TAMPERE UNIVERSITY OF TECHNOLOGY

HAO LIU

PRE-TRAINING IN CONVOLUTIONAL NEURAL NETWORKS

Master of Science Thesis

Examiner: Heikki Huttunen
Examiner and topic approved by the Faculty of Computing and Electrical Engineering
on 5th October 2016

ABSTRACT

TAMPERE UNIVERSITY OF TECHNOLOGY
Master's Degree Programme in Signal Processing
LIU, HAO: Pre-training in Convolutional Neural Networks
Master of Science Thesis, 52 pages, 16 Appendix pages
December 2016
Major: Signal Processing
Minor: Learning and Intelligent Systems
Examiner: University Lecturer Heikki Huttunen
Keywords: artificial neural network, autoencoder, pre-training, deep learning

The objective of this thesis is to study unsupervised pre-training in convolutional neural networks (CNNs) with a special kind of artificial neural network (ANN) called autoencoder. Unsupervised pre-training was proposed to solve the problem of training a deep neural network with more than one hidden layer by pre-training it in a greedy layer-wise manner. Although it started to lose its popularity after some advances in optimization and initialization methods, it is still worthwhile to study their effect on modern neural networks like CNNs.

Two new methods of applying autoencoders or their variants for pre-training a CNN are proposed in this work. One is embedding the classifier as the encoder part of an autoencoder to reduce network complexity. In a conventional pre-training method applying autoencoders, the output layer of a classifier is usually randomly initialized. However, the proposed method initializes all the layers by unsupervised pre-training. The other model applies labeled data to build a supervised variant of autoencoder.

From experiments conducted on MNIST and CIFAR datasets, unsupervised pre-training still help with improving network performance of CNNs. The embedded classifier gives compatible results to the conventional pre-training method while the supervised variant performs mostly better.

PREFACE

This thesis work was undertaken at the Computer Vision Group in Department of Signal Processing, Tampere University of Technology with financial support provided by Professor Joni-Kristian kämäräinen and Tamlink Oy.

First of all, I would like to express my sincere gratitude to my supervisor, University Lecturer Heikki Huttunen, who has been always encouraging me to explore and providing excellent guidance whenever needed, for his patient mentoring and great help during this process. I wish to thank professor Joni-Kristian Kämäräinen for helpful discussions in weekly meetings and funding support. I would like to thank Tamlink Oy for providing financial support. I am also grateful to CSC-IT Center for Science, Finland (CSC) and Tampere Center for Scientific Computing (TCSC) for providing computing resources.

I would like here to thank Department of Signal Processing and professor Irek Defee for the wonderful mentor programme. This programme helped me quite a lot getting familiar to studying at TUT.

I would also like to extend my appreciation to Computer Vision Group members and all my friends here in Finland. It is you people that makes the wonderful experience here even more fantastic.

Finally, I would like to thank my parents and sister for understanding and supporting every step I took in my life.

Hao Liu
October 2016
Tampere, Finland

CONTENTS

TERMS AND DEFINITIONS

ANN	Artificial Neural Network
BP	Back-Propagation, a common method to train an ANN
CAE	Convolutional Autoencoder
CNN	Convolutional Neural Network
DAE	Denoising Autoencoder
DBN	Deep Belief Network
DL	Deep Learning
DNN	Deep Neural Network
GPU	Graphics Processing Unit
i.i.d.	Independent and Identically Distributed, a distribution of random variables
ML	Machine Learning
MLP	Multi-Layer Perceptron
RBM	Restricted Boltzmann Machine
ReLU	Rectified Linear Unit, activation function
ROC	Receiver Operation Curve, metric to measure network performance
SGD	Stochastic Gradient Descent, optimization method
VGG	Visual Geometry Group, machine learning group at University of Oxford

1. INTRODUCTION

Machine learning (ML) is a scientific discipline solving the problem how a computer program can improve itself with experience. In a sense, machine learning techniques extract interesting representations from data and process them with regard to some objectives. Ever since 1959 when Samuel developed a computer program playing Checkers [1], which convinced himself and other people that a computer program can learn from its experience, machine learning has been drawing wider and heavier attentions.

Deep learning (DL) is one of the main aspects what machine learning field currently concentrates on. It applies Artificial Neural Networks (ANNs) that have more than one hidden layer to process data for extracting interesting features. However, it was long believed that training a deep neural network with more than two hidden layers is too difficult. The deep networks often performed no better, actually often worse, than shallow ones. But as this is a negative effect, it was seldom reported [2].

In 2006, Hinton et al. [3] proposed a greedy layer-wise unsupervised training method for Deep Belief Networks (DBNs), which significantly improved performance of deep networks. Each layer in a DBN is viewed a Restricted Boltzmann Machine (RBM), which learns probability distribution over input data. Each RBM learns data distribution from its previous layer and transfer its output to the following one. This layer-wise pre-training strategy enabled training a deep network much more easily and improved network performance significantly. Meanwhile, another artificial neural network named autoencoder was proposed, primarily for dimension reduction [4]. Autoencoders have a nice property that they try to reconstruct input data at output end, which guarantees that the intrinsic information buried in input data will go through the whole network no matter how the network looks like. This property makes it an ideal choice for pre-training an ANN similar to using a RBM by stacking them on top of each other [5]. Those pre-training techniques showed superior performance and were thus adopted by many researchers in the following years.

Unsupervised pre-training techniques provided a route for training deep networks by dividing training procedure into many stages and only training a shallow one at each stage. However, they did not solve the problem that it is difficult to train a deep network directly from scratch. In 2010, Glorot et al. [6] studied the saturation

problem in deep neural networks and proposed a novel initialization method for deep networks. The proposed initialization method, often named Xavier Initialization or Glorot Initialization by other researchers, made training a deep network from scratch possible. One year later, a new activation function, which is called Rectified Linear Unit (ReLU) [7] was proposed to accelerate training as well as reduce saturation. As it has linearity for activation values greater than 0, it does not suffer from saturation problem or vanishing gradients. All those advantages made it much easier to train a deep network from scratch. New techniques mentioned above, together with the fact that there are more and more large datasets like ImageNet [8] coming available, made unsupervised pre-training less and less necessary. It gets unpopular to apply pre-training techniques.

However, in a recent work by Paine et al. [9], experimental results show that pre-training still helps when there are more unlabeled data available than labeled ones. Consider the fact that obtaining unlabeled data is significantly cheaper than getting labeled ones, it is still worthwhile to study pre-training techniques even after we have those new methods like Xavier Initialization, ReLU and so forth. Furthermore, unsupervised learning is considered [10] to be one of the most promising machine learning techniques in the future. As an important branch and good starting point, unsupervised pre-training deserves more attention.

In this thesis work, we study whether and how pre-training with autoencoders help with modern networks like Convolutional Neural Networks (CNNs) and propose new ways of using autoencoders for initialization.

The remaining parts of this thesis are organized as follows. Chapter 2 and 3 explain some ideas to help readers understand the topic discussed later. Chapter 2 focuses on different topology of ANNs. Several techniques applied to ANNs are the main interests in Chapter 3. Chapter 4 presents the experiments in detail and discusses the results.

2. METHODS

This chapter introduces structures of ANNs, CNNs and autoencoders in detailed manner. As the basic component, neurons will also be discussed at the beginning of this chapter.

2.1 Artificial Neural Network

This section discusses what an ANN is. It starts with the computing units of an ANN, which are called *neurons*, and activation functions related to them. History of ANNs is introduced later in this section.

2.1.1 Neuron

Neurons are the fundamental computing units in an artificial neural network. Typically, an ANN will contain numerous neurons in a layer-sequential order. The neurons take input from source data or activation values from other neurons and compute single output based on connecting weights. By stacking layers that contain those computing units, a network can show impressive representative ability. The basic structure of a neuron can be seen in Figure 2.1.

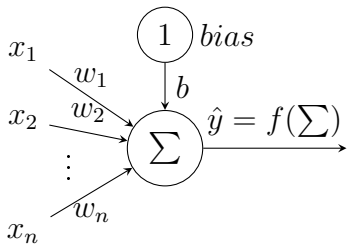

Figure 2.1: Basic structure of a neuron

As shown in the figure, signals are connected to the neuron through different connections and added together according to the weight for that connection. That is, the output of a neuron can be calculated according to

$$\hat{y} = f\left(\sum_{i=1}^{n} w_i x_i + b\right) = f\left(w_1 x_1 + w_2 x_2 + \cdots + w_n x_n + b\right), \tag{2.1}$$

where b is the bias of this neuron. For simplicity, the bias can be seen as a special

input signal to the neuron which always takes integer 1 as its value. In this case, the output of a neuron can be denoted as

$$\hat{y} = f\left(\sum_{i=1}^{n} w_i x_i + b\right) = f\left(\vec{w}^T \vec{x}\right), \tag{2.2}$$

where $\vec{x} = [x_1, x_2, \cdots, x_n, 1] \in \mathbb{R}^{n+1}$ represents input values connected to this neuron and $\vec{w} = [w_1, w_2, \cdots, w_n, b] \in \mathbb{R}^{n+1}$ is the weight vector. The function $f(\cdot)$ in the figure means an activation function, which maps the weighted sum to a value in a certain range. Theoretically, any function taking a single input and mapping it to a deterministic value can be used as an activation function. However, as we will see below, no mater how many hidden layers there are in a neural network, if all the neurons map weighted sum linearly (e.g. $f(z) = \alpha z + \beta$, where α and β are constant coefficients), the whole network can be replaced with a two-layer input-output model.

2.1.2 Nonlinear activation

Given a neural network, consider its l^{th} layer whose weights for its i^{th} neuron are denoted by $\mathbf{W}_i{}^{(l)} \in \mathbf{R}^N$, where N is the amount of weights this neuron has. Because a neuron is fully-connected to all the neurons in previous layer, $\mathbf{W}_i{}^{(l)}$ here is actually a vector. For simplicity, we take identity function $f(z) = z$ as an example. The output value of this neuron, denoted by $\mathbf{y}_i{}^{(l)}$ will be

$$\mathbf{y}_i{}^{(l)} = [\mathbf{W}_i{}^{(l)}]^T \mathbf{y}^{(l-1)}, \tag{2.3}$$

where $\mathbf{y}^{(l-1)}$ is output from its previous layer, or input data for the first layer. Therefore, output from l^{th} layer will be

$$\mathbf{y}^{(l)} = [\mathbf{W}^{(l)}]^T \mathbf{y}^{(l-1)}. \tag{2.4}$$

At $(l+1)^{th}$ layer, we will similarly have

$$\mathbf{y}^{(l+1)} = [\mathbf{W}^{(l+1)}]^T \mathbf{y}^{(l)} = [\mathbf{W}^{(l+1)}]^T [\mathbf{W}^{(l)}]^T \mathbf{y}^{(l-1)}. \tag{2.5}$$

So finally at the output end (denoted as L^{th} layer), it will be

$$\mathbf{y}^{(L)} = [\mathbf{W}^{(L)}]^T [\mathbf{W}^{(L-1)}]^T \cdots [\mathbf{W}^{(1)}]^T \mathbf{x} = \mathbf{W}^T \mathbf{x}, \tag{2.6}$$

where $\vec{x}$ is input data and $\mathbf{W}$ represents the product of weights along all layers. In another word, we cannot benefit from stacking layers together to get a more representative model if the neurons have only identity activation functions. For any

other linear functions $f(z) = \alpha z + \beta$, it is quite the same except that there will be one more element in each $\mathbf{W}_i{}^{(l)}$ representing β and one more element in each $\mathbf{y}_i{}^{(l)}$ with value 1.

Nevertheless, things are different for nonlinear activation functions. It has been proven [11] that a two-layer computing network with sigmoidal activation can form a universal function approximator, which shows the superiority of nonlinear activations. Moreover, a network can learn much complicated representations of data with same amount of neurons if there are more layers with nonlinear activation functions [12]. Therefore, nonlinear activation like sigmoidal function has become a typical choice in neural networks.

Although sigmoidal function is the classic choice for bring nonlinearity in ANNs, it is not the only option. In a sense, any single-input-single-output mapping can be used as an activation function. There are several other commonly used activation functions, including hyperbolic tangent (tanh), softsign, rectified linear unit (ReLU) [7] and softmax [13].

Logistic function

Logistic function, which is defined as

$$\delta(x) = \frac{1}{1 + e^{-x}}, \tag{2.7}$$

will map its input value to fit in range (0, 1) and thus is commonly used for classification tasks with its output viewed as the probability value. It is usually referred as *sigmoid* function.

Hyperbolic tangent

$$tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}} \tag{2.8}$$

Hyperbolic tangent function is often called *tanh* for short. As defined in Equation 2.8 , hyperbolic tangent function maps values in $(-\infty, \infty)$ to (-1, 1).

Softsign

$$softsign(x) = \frac{x}{1 + |x|} \tag{2.9}$$

Softsign activation is defined in Equation 2.9. Although it also maps its input to (-1, 1) as hyperbolic tangent does and they have similar shape as shown in Figure 2.2, it reaches its asymptotic much slower than *tanh*.

ReLU

Rectified Linear Unit (ReLU) is, as its name indicates, an activation function which encourages competition between weighted sum with 0 (e.g. rectified values), it is defined as

$$\alpha(x) = \begin{cases} 0, & x < 0 \\ x, & x \geq 0. \end{cases} \tag{2.10}$$

It introduces non-linearity by forcing negative values to be zero. As it has unit gradient for positive values, it do not suffer from vanishing gradient problem [7]. Meanwhile, it suffers much less saturation problem than sigmoid, which have made it a popular choice in ANN field and several variants of this function has been proposed [14; 15; 16; 17; 18]. Based on a summary [19], ReLU is the most popular choice in deep learning field in 2015.

Figure 2.2 shows how those activation functions look like. Although tanh looks similar to sigmoid, it approaches its asymptotic much slower.

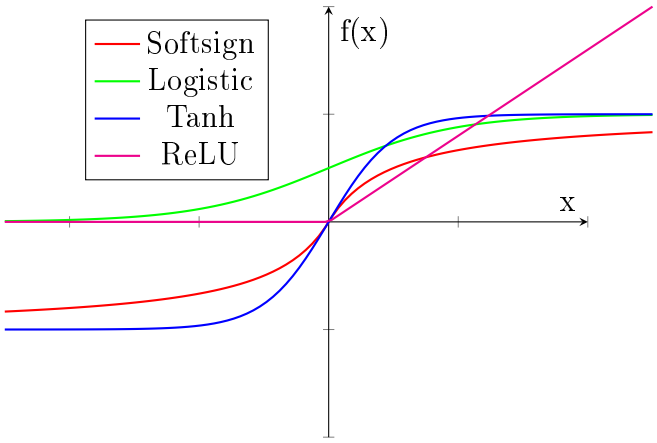

Figure 2.2: Commonly used activation functions

Softmax

Softmax [13] is another widely used activation function. Different from all those functions mentioned above, there is not a fixed curve to illustrate this function because it is data-related. Given a sequence $x = [x_1, x_2, \cdots, x_n] \in \mathbb{R}^n$, its activation value for each element, x_i, is defined as

$$\text{softmax}(x_i) = \frac{x_i}{\sum_{k=1}^{n} x_k}. \tag{2.11}$$

It has the property that the resulting values sum up to 1, which makes it a popular choice for normalizing output values of multiclass classifiers.

2.1.3 History of ANN

Although ANNs began to draw heavy attention and create state-of-art results, thanks to its impressive representative ability, from just more than ten years ago, the history of ANN could date back to half centuries earlier.

Before 1940s, researches related to neural networks concerned only nervous systems in animals or human beings. At that point, the word "neural network" or "neural net" was used mainly for describing neural systems. In 1943, McCulloch and Pitts [20] proposed a computational model for neural networks based on some mathematical rules. Starting from this work, researchers began to explore neural networks for their possible applications in artificial intelligence.

In 1958, Rosenblatt [21] studied neurons and tried mathematically emulating them. A mathematical model named *perceptron* was proposed and showed its learning ability. This has been well recognized as one of the very first ANNs.

Eleven years later, Minsky and Papert [22] published their work, where two disadvantages of ANNs were issued. One was that a perceptron cannot learn representations for exclusive-or. The other one was that computers were not powerful enough to deal with heavy calculations required by ANNs. This work discouraged researchers and other people, slowed down progress of ANNs.

From 1970s to 1980s, back-propagation (BP), which is abbreviation for "backward propagation of errors", was proposed and finally applied to ANNs on computers [23; 24; 25; 26]. As an optimization method, it solved the exclusive-or problem and made it possible to train an ANN efficiently.

Although BP was proposed and optimized for computer programming, it was not practical to train a network with more multiple layers in a tolerable time before the computers became fast enough. Fortunately, computational power of computers had kept increasing significantly and researchers began to use GPUs for computing. From 2005 when GPUs were used for LU decomposition [27], we have seen the trend that GPU gets more and more faster than CPUs for general computing.

Thanks to the fast development of faster computers, especially faster GPUs, tensive experiments have been conducted and a huge amount algorithms and methods have been illustrated with those experiments during the past decade. For regularization, Dropout [28] was proposed to regularize an ANN by randomly dropping some neurons. DropConnect [29], on the other hand, drops connections between neurons in a random order. Adagrad [30] and Adadelta [31] were proposed to better optimize networks during training progress. With a good initialization method like Xavier Initialization [6] or Layer-Sequential Unit Variance [32], it gets easier to optimize a network more efficiently and make it generalize better. All those techniques, along with many other remarkable work, make it possible to train very deep networks. In

2014, a by-then incredibly deep network proposed by VGG team [33] settled impressive results on ImageNet Challenge 2014 [8]. One year later, a 152-layer network which applied the idea of residual learning [34], proposed by Microsoft Research Asia, won several important competitions with state-of-art results, having created a new record for network depth and network performance. The same idea can even be applied to unbelievably deep networks with more than 1000 layers.

When there are more than one hidden layer in a network, we typically refer it as a deep neural network (DNN). Applying DNNs to solve computer vision problems or any other tasks is oftentimes names deep learning (DP). As shown in previous paragraph, DNNs have achieved impressive results during past few years and is now the most popular topic in machine learning filed.

2.2 Convolutional Neural Network

Most remarkable networks mentioned in Section 2.1.3 are from a branch of modern neural networks, which is called Convolutional Neural Network (CNN). It was inspired by a neuroscience research conducted half centuries ago.

In 1968, Hubel and Wiesel [35] discovered that neurons in visual cortex of cats respond to synaptic signals selectively. Neurons in visual cortex are arranged in a pyramid-like order and a neuron in a high level will most accept signals from a small receptive filed in its previous level. Those receptive fields overlap with each other and are connected to different neurons. Inspired by this work, Fukuyama proposed neocognitron [36] in 1979. The network is arranged in a layer-sequential order and it has multiple layers. Although a different optimization method is applied in the network, which differs from back-propagation based algorithms used in modern networks, the ideas of convolution and pooling are still the fundamental of modern convolutional networks. The convolutional neural network applied to zip code recognition [37] was the very first practical application utilizing CNNs. Since then, CNNs have been drawing heavy attentions. Especially from 2012 when a very large scale CNN [38] showed an amazingly exciting performance on ImageNet2012 [8] benchmark with an at least 9.7% lower error rate than any other methods, CNNs have been gaining higher popularity.

Typically, a convolutional neural network contains one or more convolution layers. Different from a conventional fully-connected neural network discussed in Section 2.1, a neuron in a convolution layer is connected to only part of neurons in its previous layer. The name "convolution layer" comes from the fact that the way how output of a convolution layer is calculated can be described by a mathematical operation called convolution.

Mathematically, convolution is an operation on two functions which yields a third one. It is a particular case of integral transform. Given two functions f and g, the

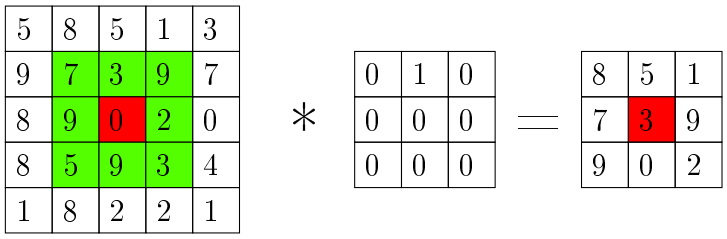

Figure 2.3: An example for convolution operation in a CNN. The red rectangle is currently where convolution occurs and blue region is the "valid area" for convolution operation.

convolution of them, which is denoted by $f * g$, can be defined as

$$(f * g)(t) = \int_{-\infty}^{\infty} f(\tau)g(t-\tau)d\tau. \tag{2.12}$$

For two-dimensional data like an image $I(x, y)$, its convolution with another 2D function $f(x, y)$ is defined as

$$(I * f)(x, u) = \sum_{u} \sum_{v} I(x-u, y-v)f(u, v), \tag{2.13}$$

which can be viewed as the dot product of $I(x, y)$ and $f(x, y)$. The latter one is called convolution kernel and it needs to be rotated for 180° based on its center before the element-wise multiplication required by dot product. In CNN implementations, dot product is usually calculated without rotating the kernel. Figure 2.3 illustrates how convolution in CNNs is calculated. This kind of convolution operation is applied to overlapped regions and each generated matrix will be referred as a feature map. This idea is named "weight sharing" because all the local areas in the input data are convolved with same kernels and their weights stay unchanged before applying to all the local regions.

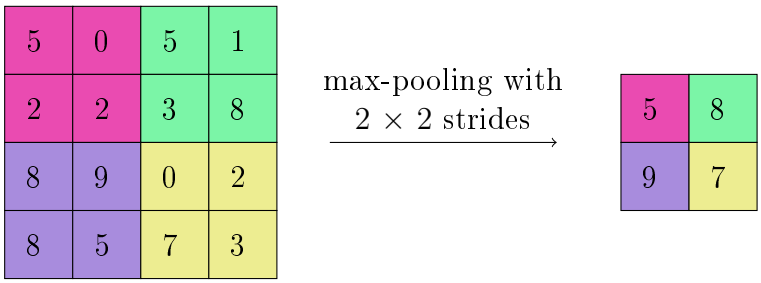

Figure 2.4: Example of max-pooling with 2×2 pooling region. The word "stride" means how many elements the pooling region moves along an axis. Thus in this case, it moves 2 elements along each axis.

A convolution layer in a CNN is oftentimes followed by a pooling layer. This layer is inspired by the selective respond fact of visual neurons [35] as discussed at the beginning of this section. Only a single value, most times the maximum one or the average of all the elements in a local region, is preserved after pooling. Therefore, a pooling layer will reduce feature map size in most cases. Figure 2.4 gives an example on how max-pooling works with input data.

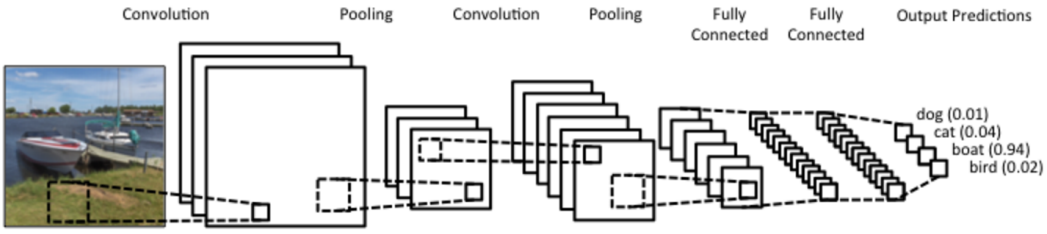

Figure 2.5: Convolution neural network for image classification[1].

Typically, a convolutional neural network contains one or more convolution layers. As shown in Figure 2.5, convolution kernels are applied to input data to extract feature maps. Those feature maps are then pooled and fed to the following convolutional layer. After another pooling stage, the feature maps are flattened into a vector, followed by fully-connected layers. At the output, it will yield probability estimation normalized by softmax activation function. Of course, the fully-connected layers are not necessary in CNNs. For instance, there has been fully-convolutional networks [39; 40] where only convolution and pooling layers are applied to build highly representative networks.

Compared with conventional feed-forward neural networks, CNNs has several advantages. First of all, it preserves correlations among neighbor elements, or spatial structure for another word, of data by convolution (actually cross-correlation as it does not rotate the kernel) operation, which has made it a popular choice for sequential data and images. Secondly, it reduces complexity of an ANN significantly especially for tensive data. consider a network whose input data is a single channel 128×128 image. If we add a fully-connected layer with 300 neurons directly to its input, there will be $128 \times 128 \times 300 = 4915.2$K parameters (weights) for this layer. However, if we first apply four convolution layers with 32 3×3 kernels, each followed by a 2×2 pooling layer, the total amount of weights for this fully-connected layer and all convolutional layers will be only 373.248K. With modern techniques for initialization and optimization mentioned in Section 2.1.3, we can easily stack more convolutional layers to further reduce complexity of networks.

[1]This image is taken from *http://www.wildml.com/2015/11/understanding-convolutional-neural-networks-for-nlp/*, retrieved on 18 Oct, 2016

Nevertheless, CNNs also have their drawbacks. The biggest problem of a convolutional neural network is that convolution is an expensive operation. Without optimized GPU implementation, it will be rather time-consuming to train a CNN, especially for big data.

2.3 Autoncoder

Autoencoder is a special kind of neural network which tries to reconstruct input data at its output. It is optimized to minimize the discrepancy between its input and output. This may seem trivial at the first glare as there is not much new information from its output. However, if the layer which has smallest output dimensionality in an autoencoder has fewer neurons than input layer, it will be forced to learn data representation at a lower dimensionality. Intuitively, if input data can go through the network and get reconstructed at output, information buried in those data will also be preserved by each of its layers. In this case, the bottleneck layer which has smallest output dimensionality will be able to extract a low-dimensional representation from input data. Therefore, it can be applied for dimension reduction and this is actually one of the main purposes it was proposed for [4]. Consider data shown in Figure 2.6, digits in the first row are presented by 28×28 gray-scale images. After passing an autoencoder with bottleneck dimensionality 50, they are reconstructed as shown in the second row.

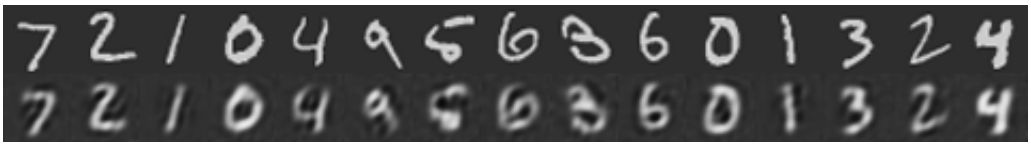

Figure 2.6: Original (first row) and reconstructed (second row) image of an autoencoder with input dimensionality 784 and bottleneck dimensionality 50.

As shown above, although reconstructed images are blurred, they are still recognizable to human beings. That is to say, the most interesting information carried by input data is able to reach output end. In this network, we are able to get a 50D representation for original 784D data.

Figure 2.7 gives a basic idea how an autoencoder looks like. The hidden layer which has lowest dimensionality is usually referred as bottleneck layer or code layer. Forward pass from input to bottleneck layer forms an encoder (e.g. encode data as a low-dimensional representation at bottleneck layer) and the other parts comprise a decoder (e.g. decode representation saved in bottleneck layer). For simplicity, there is only one hidden layer in this figure thus it is the bottleneck layer. However, an autoencoder could have any number of hidden layers as needed. Also, the network topology does not necessarily needs to be symmetrical as shown in Figure 2.7. As

long as it aims at reconstructing input data at its output, it can be referred as an autoencoder.

Besides reconstructing input signal itself, autoencoders can also be use for other purposes. For examples, denoising autoencoders (DAEs) [41] try to restore clear data from blurred source. When training a DAE, noise is added to the clear source but not target output. After optimization, autoencoders will be able to learn representations from noisy source as well as from clear input as they are able to learn only repeated patterns while noise is highly random.

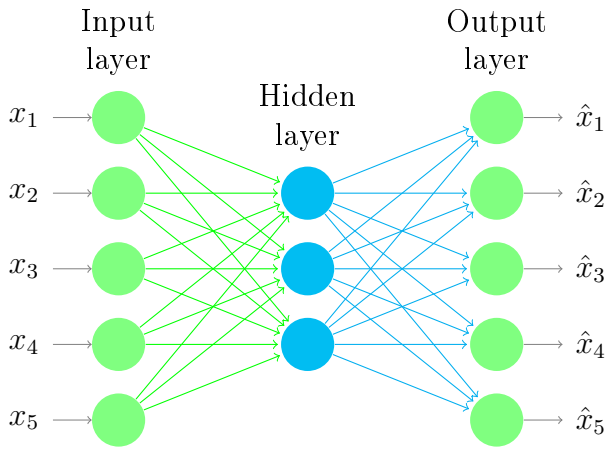

Figure 2.7: Example of a simple autoencoder that attempts to reconstruct 5-dimensional data from 3-dimensional representation saved in the hidden layer

With advantages of convolutional layers discussed in previous section, autoencoders can also contain convolutional layers. This kind of autoencoders is usually called a convolutional autoencoder (CAE). A neural network [42] staking several CAEs achieved superior performance on two computer vision benchmarks.

Since it was proposed, autoencoder has been widely applied for dimension reduction [4; 43], learning generative models of data [44; 45], data denoising [46; 47] and transfer learning [48; 49; 50]. Zhang et al. [51] studied imbalanced data classification problems and applied a denoising autoencoder to restore blurred images for over-sampling. Li et al. [52] used autoencoder to extract feature from YCbCr space and achieved better performance with same down-sampling ratio than other methods. For visual speech recognition tasks, Petrids et al. [53] extracted bottleneck features with autoencoders and added new layers on top of feature vectors to form a more accurate classifier.

There are also variants of autoencoders. Wang et al. [54] used each pixel to reconstruct a set of output values instead of just recovering itself. Results show that by combining different algorithms like LDA with this idea, the variants of autoencoders can outperform their conventional counterparts. Makhzani et al. [55] proposed a novel method to sparsify an autoencoder. Different from combining

individual nonlinearities together, the so-called K-sparse autoencoder introduces sparsity by preserving only K largest activations for the code layer and setting everything else to be zero. That's the only nonlinearity in a shallow autoencoder, everywhere else is just linear activation. Results on digit classification and object recognition tasks show that this method can give compatible or even better results than by-then-state-of-art results. Another variant, also proposed by the same authors [56], keeps K percent largest values and sets the left to be zero for each feature map. This variant, which was named Winner-take-all autoencoder, also gives very good performance as it prevents network from recording the original input to several certain feature maps. Stacked What-Where autoencoder [57] is another variant of conventional autoencoders. It assumes that, in order to restore input data, not only the pooled value ("what" information), but also the position where it is pooled from ("where" information) is. By applying both "what" and "Where" information, it showed amazingly good reconstruction results. All those methods mentioned above apply a conventional strategy shown in Equation 2.1, e.g. a bias is added to weighted sum of signals connected to a neuron. Konda et al. [17] proposed a novel method to train an autoencoder without bias. This strategy, together with a new activation function, enabled autoencoders to learn data representations with extremely high dimensionality where conventional autoencoders typically fail.

For other researches, Li et al. [58] studied effect of whitening transformation on pooling operations in convolutional autoencoders and found that average pooling suits data applied whitening transformation while max pooling gives more desirable results for data without such processing. Lin et al. [59] used zero-biased autoencoders [17] for unsupervised pre-training and achieved performance close to what was achieved by CNNs.

3. IMPLEMENTATIONS

This chapter discusses several important algorithms related to ANNs, mainly concerning how network performance can be assessed, how network parameters can be initialized and optimized and how conventional methods apply autoencoders for unsupervised training. The proposed methods will be introduced later in this chapter.

3.1 Performance assessment

This section gives introduction about how network performance can be assessed for classification tasks.

3.1.1 Training/test split

Training/test split divides data into subsets. It is a common way to test a model which is applied for prediction tasks like classification or regression. The training data is used for optimizing the network while test set is used to measure how well a model performs for *unknown* samples. If a network can perform well only on training set but give much worse on test set, it is said to be over-fitted thus cannot generalize well. It is common to further divided training set into two parts, one of which is used for validation during training procedure to prevent over-fitting.

3.1.2 Classification Accuracy

Classification accuracy is defined as

$$\text{Acc} = \frac{\text{NumberOfCorrectlyClassifiedSamples}}{\text{NumberOfTotalSamples}} \times 100\%. \tag{3.1}$$

This is one of the most commonly used metrics to evaluate a network with regard to a classification task. However, for unbalanced data, it is not recommended to use only this metric. Consider an example dataset where there are 9990 positive samples and 10 negative ones. Even if a model makes no prediction and simply classifies every samples to be positive, it still has a 99.9% accuracy.

3.1.3 Confusion Matrix

Confusion matrix is a commonly used measurement for classification tasks, it lists how many samples are correctly and incorrectly predicted.

As show in Table 3.1, when a positive sample is predicted correctly, it contributes to true positive (TP), otherwise it will be marked false negative (FN) as the negative prediction is wrong. Similarly, when a negative sample is predicted to be negative, it contributes to true negative (NG), false negative (FG) otherwise. Based on this idea, there exist two metrics which are commonly applied, recall and precision.

Table 3.1: Example of a confusion matrix, TP means True Positive, FN means False Negative, TN means True Negative and FP means False Negative.

Real\Predicted	True	False
True	TP	FN
False	FP	TN

Recall is a metric which measures how many positive samples are detected among all the samples. In the confusion matrix shown above, total number of positive samples is $TP+FN$, while the detected positive samples are TP. As a result, recall can be calculated as

$$\text{Recall} = \frac{TP}{TP + FN} \tag{3.2}$$

Precision, on the other hand, measures how accurate the model detects positive samples. In Table 3.1, there are $TP + FP$ samples which are marked as positive instances. However, only TP samples are really positive. Therefore, precision will be measured as

$$\text{Precision} = \frac{TP}{TP + FP} \tag{3.3}$$

Neither single recall nor precision can predict a model well enough. Consider a classifier which classifies all the samples to be positive. In this case, the confusion matrix may look similar to Talbe 3.2.

Table 3.2: Example of a confusion matrix, TP means True Positive, FN means False Negative, TN means True Negative and FP means False Negative.

Real\Predicted	True	False
True	TP=2	FN=0
False	FP=998	TN=0

In this case, the model will have a highest recall with $recall = TP/(TP + FN) =$

100%. Nevertheless, the precision is only 0.2%. It has much more false samples than true ones. On the other hand, single precision is not enough to describe the model either. Consider another example shown below with a confusion matrix as shown in Table 3.3.

Table 3.3: Another example of confusion matrix

Real\Predicted	True	False
True	TP=1	FN=999
False	FP=0	TN=0

In this example, the model classifies almost all the samples to be negative. As a result, it has a precision 100% but only 0.1% recall. It misclassifies 999 samples out of 1000 but still have a nice precision.

To solve this problem, another metric is defined to describe the relationship between true positive rate and false positive rate, which is called receiver operating curve (ROC).

ROC describes how true positive rate varies with regard to false positive rate. The perfect curve should be the one at top-left corner in Figure 3.1, e.g. the true positive rate is already near 100% when false positive rate is still near zero.

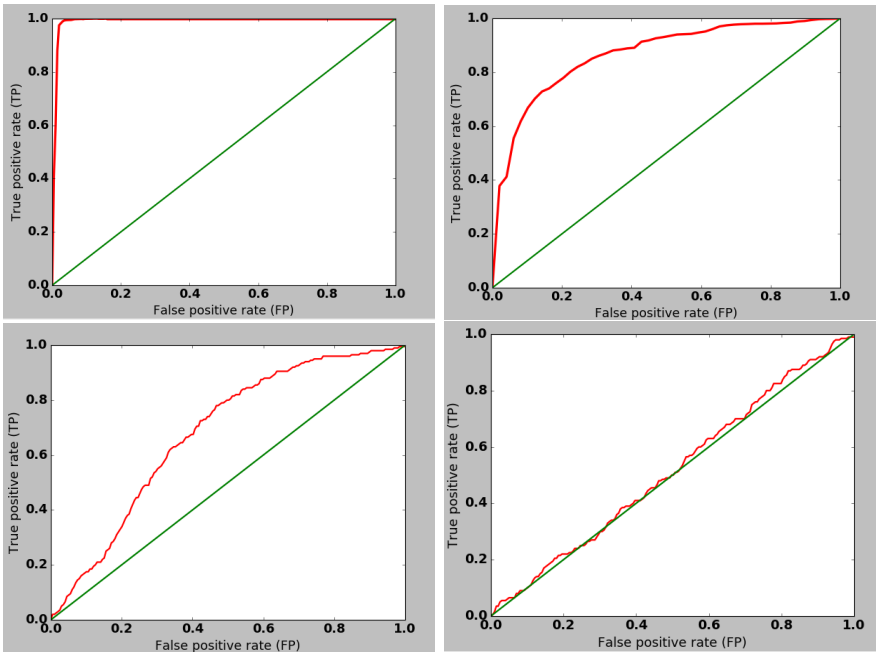

Figure 3.1: An example showing receiver operating curve. Top left one is a near-perfect classifier and the one on its right side is reasonably good. Howerer, performance of those shown in the second row is quite poor. Especially the right bottom one is no good than purely random separation (green line in those figures).

To further interpret ROC, one can use area under the ROC (AUROC, or more commonly AUC for abbreviation) as a measurement [60]. Any model performing better than random guessing should have an AUC value higher than 0.5.

3.1.4 Other metrics

There are also some other metrics for evaluating a neural network. For instance, in one *epoch*, all samples are used once to updating parameters. Number of epochs needed for convergence can be used as a measurement to describe how efficiently a neural network uses training data for optimization.

3.2 Loss function

In machine learning, an optimization algorithm seeks to minimize or maximize a mathematical function mapping one or more values into a real number. This function to minimize or maximize is called loss function. Different from metrics discussed above, a loss function needs to be differentiable. The most commonly used loss functions include mean squared error (MSE) and cross-entropy.

3.2.1 MSE

MSE measures average value of squared distance between two vectors. It is widely applied in regression tasks. Mathematically, MSE of two vectors $\vec{x} = [x_1, x_2, \cdots, x_n] \in \mathbb{R}^n$ and $\vec{y} = [y_1, y_2, \cdots, y_n] \in \mathbb{R}^n$ is defined as

$$\mathrm{MSE}(\vec{x}, \vec{y}) = \frac{1}{n} \sum_{i=1}^{n} (x_i - y_i)^2 \tag{3.4}$$

3.2.2 Cross-entropy

Cross-entropy is commonly used for multi-class classification tasks. It can give a even better idea about how well the network classifies samples than classification accuracy metric in some cases.

Assume we have a binary classification problem and two individual networks whose output values are described in Table 3.4.

Table 3.4: Output of two different networks

network output		ground truth		result
0.95	0.05	1	0	Correct
0.90	0.10	1	0	Correct
0.02	0.98	0	1	Correct
0.41	0.59	1	0	Wrong

network output		ground truth		result
0.55	0.45	1	0	Correct
0.70	0.30	1	0	Correct
0.37	0.63	0	1	Correct
0.22	0.78	1	0	Wrong

As we can see, both networks have classification accuracy Acc = 75% as they classify 3 out of 4 instances correctly. However, the left one is better than its right counterpart intuitively because if it is correct, it has high confidence while it has lower confidence when it misclassifies. So here comes cross-entropy to distinguish those different networks better especially when they have similar classification accuracy.

Mathematically, cross-entropy between two different discrete probability distribution over same dataset D, denoted by p and q, is defined as

$$Cros(p, q) = -\sum_{x \in D} p(x) \ln q(x), \tag{3.5}$$

It measures the distance between data distribution predicted by a model and the real distribution. Smaller cross-entropy gives lower discrepancy thus better performance. More generally, cross-entropy error function for an ANN is proposed in a work [61] as follows. For n-element vectors, target values $\vec{t} = [t_1, t_2, \cdots, t_n] \in \mathbb{R}^n$ and network output $\vec{y} = [y_1, y_2, \cdots, y_n] \in \mathbb{R}^n$, cross-entropy over those two vectors can be calculated as

$$E_n = -\frac{1}{n} \sum_{k=1}^{n} [t_k \ln y_k + (1 - t_k) \ln (1 - y_k)], \tag{3.6}$$

where t_k is the target value and y_k is network output. For the example given in Table 3.4, Cross-entropy for the two nets are 0.1160 and 0.3182 respectively. As we can see, even when the networks have identical classification accuracy, we can still compare them using cross-entropy error.

3.3 Optimization

The procedure of training a neural network is updating its weights and trying to find an optimal point in parameter space which can minimize loss function over training data. Gradient based methods like Stochastic Gradient Descent (SGD) are the common choice for optimizing the parameters.

3.3.1 Stochastic Gradient Descent

Stochastic Gradient Descent(SGD) is an optimization method widely applied in ANN field. It is a stochastic approximation of standard gradient descent method and can converge to a local minimum faster than the standard one. It guarantees to converge [62; 63] no matter data is linearly separable or not.

To help understanding how SGD works, assume we are standing on a hill (e.g. the highest point in figure 3.2) and we want to down the mountain with a path as short as possible. No GPS or a map or any electricity devices are available and it is

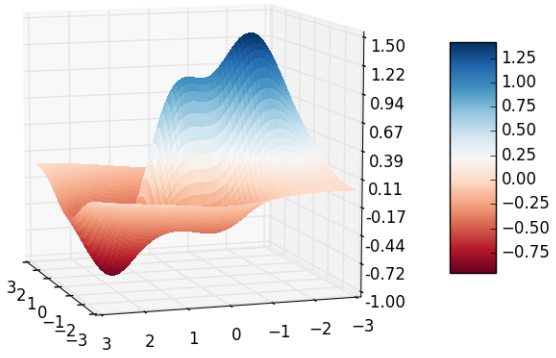

Figure 3.2: An examples about a two-dimensional case, the two axes represent two variable and the surface gives an idea how large the error is wrt. those two axes.

so foggy that we can barely see anything farther than half meters away. How can we find a right path to help us down the mountain most quickly? One of the intuitive way will be looking around from where we stand and trying to find a direction the altitude drops fastest. Because in that direction, it is most possible to be the shortest path. Of course no one can promise it to be shortest as it might getting much smoother after just from 2 meters away. However, it is more possible to be a shorter path than those directions where gradient is smaller as we can go down most within our eyesight along that direction. After we move to the new position, we will be able to see another half meters from the new point and go down a little bit in the same way until we reach a basin and we can only go up from that point. This is what gradient descent does: find a direction which the error drops fastest (e.g. largest gradient) and go little further to that direction and repeat it again until a local minimum is reached.

Consider training set $X = \{\vec{x}_1, \vec{x}_2, \cdots, \vec{x}_m\}$ as an examples. The target output for X is denoted by $Y = \{\vec{y}_1, \vec{y}_2, \cdots, \vec{y}_m\}$. A model whose output can be denoted by $\hat{y} = f(\vec{w}, \vec{x})$ will have empirical risk [64] calculated as

$$e(\vec{w}) = \frac{1}{m} \sum_{k=1}^{m} e(\hat{y}_k, \vec{y}_k), \tag{3.7}$$

where $e(\hat{y}_i, \vec{y}_i)$ is a function describing discrepancy between actual output and target value, mean squared error (MSE) between two vectors for example. The vector $\vec{w}$ consists parameters to optimize.

In general, given error function denoted by 3.7, we can reduce error by going op-

posite its gradient direction and repeat iteration until convergence. For i^{th} iteration, weights $\vec{w}$ are updated by

$$\vec{w} \leftarrow \vec{w} + \Delta\vec{w}_i, \tag{3.8}$$

where $\Delta\vec{w}$ is the adjustment of weights for i^{th} updating and it can be calculated by

$$\begin{aligned} \Delta\vec{w}_i &= -\eta \bigtriangledown e(\vec{w}) \\ &= -\eta \frac{1}{m} \sum_{k=1}^{m} \bigtriangledown e(\hat{y}_k, \vec{y}_k). \end{aligned} \tag{3.9}$$

In Equation 3.9, η is called learning rate, which is a small value controlling how far we go in each iteration. Therefore, weights are updated as

$$\vec{w} \leftarrow \vec{w} - \eta \frac{1}{m} \sum_{k=1}^{m} \bigtriangledown e(\hat{y}_k, \vec{y}_k) \tag{3.10}$$

over every iteration. Choosing proper η is crucial as number of iterations needed to converge for a small learning rate may be significantly large. On the other hand, a large learning rate can easily lead the model to skip optimal points, preventing it from converging for extreme cases.

Discussions above shows how standard gradient descent optimizes a model. As indicated in Equation 3.10, prediction error for all samples needs to be added together before updating weights, which is an expensive work especially for large datasets. For online learning problems, it is even impossible to applying summing over infinite samples. To speed up converging, SGD, which is a stochastic variant of standard gradient descent, is preferred in real problems. It is a good approximation for standard gradient descent with much faster convergence speed [65]. Instead of summing prediction error over whole dataset, SGD updates weights based on each individual sample. That is, weights are updated with

$$\vec{w} \leftarrow \vec{w} - \eta \bigtriangledown e(\hat{y}_i, \vec{y}_i) \tag{3.11}$$

for $i = 1, 2, \cdots, m$. As it uses individual samples for updating weights, which brings high randomness, it is more likely to escape from poor local minima. However, it might also leave global optimal point due to same reason.

In modern deep learning frameworks like Theano [66], TensorFlow [67] etc., SGD is often implemented as Mini-Batch Gradient Descent (MBGD). Different standard gradient descent and SGD, it updates parameters by summing error over certain amount of samples, which benefits from both low randomness and fast convergence

provided suitable batch size b. Its weights are updated by

$$\vec{w} \leftarrow \vec{w} - \eta \frac{1}{b} \sum_{k=1}^{b} \bigtriangledown e(\hat{y}_k, \vec{y}_k). \tag{3.12}$$

Usually, we can use a larger learning rate at the beginning to approach a local minimum faster and use a smaller value when it is close to a local optimal point. For example, in this thesis work, learning rate at i^{th} update will be

$$\eta_i = \frac{\eta_{i-1}}{1 + i \cdot \beta}, \tag{3.13}$$

where β is a small value controls how much learning rate decays for each update. As a result, given initial learning rate η_0, how weights are updated at i^{th} updating will be

$$\vec{w} \leftarrow \vec{w} - \eta_0 (\prod_{l=0}^{i-1} \frac{1}{1 + l \cdot \beta}) \frac{1}{b} \sum_{k=1}^{b} \bigtriangledown e(\hat{y}_k, \vec{y}_k) \tag{3.14}$$

for MBGD.

To accelerate convergence, a momentum [26] can be added to the updating of weights. For standard gradient descent, it will be

$$\Delta \vec{w}_i = -\eta \bigtriangledown e(\vec{w}) + \alpha \Delta \vec{w}_{i-1}, \tag{3.15}$$

where $\Delta \vec{w}_i$ is updated amount for i^{th} iteration, $\Delta \vec{w}_{i-1}$ is the updated value for previous updating and α is a constant typically ranging in $[0, 1]$. The main idea behind momentum is that, if a gradient consistently points to similar directions, it helps going further along that direction for each step to accelerate converging. The updating rule for SGD on i^{th} updating with momentum is

$$\vec{w} \leftarrow \vec{w} + \Delta \vec{w}_i \tag{3.16}$$

with

$$\Delta \vec{w}_i = -\eta \bigtriangledown e(\hat{y}_i, \vec{y}_i) + \alpha \Delta \vec{w}_{i-1} \tag{3.17}$$

For the purpose of accelerating convergence, Nesterov Accelerated Gradient [68] can also be applied. As shown in Equation 3.17, classical methods correct updates with momentum from last iteration before moving to a new position. On the contrary, Nesterov Accelerated Gradient first moves to a new position based on previous momentum and then correct it according to the new point. Mathematically, the updating for i^{th} iteration can be denoted as

$$\Delta \vec{w}_i = -\eta \bigtriangledown e(\vec{w}_{i-1}) + \alpha \Delta \vec{w}_{i-1}. \tag{3.18}$$

Thus the weights are updated at i^{th} updating by

$$\vec{w}_i \leftarrow \vec{w}_{i-1} + \alpha\Delta\vec{w}_i - \eta \bigtriangledown e(\vec{w}_i). \tag{3.19}$$

3.3.2 Adadelta

Adadelta [31] is another gradient-based optimization method. Different from SGD, it does not require manually-selected learning rate.

Adadelta is inspired by Adagrad [30], which has been found to improve robustness of SGD [69]. Instead of applying a pre-defined learning rate decay, Adagrad reduces learning rate based on samples fed into the model during optimization. Its updating rule is defined as

$$\Delta\vec{w}_i = -\frac{\eta_0}{\sqrt{\sum_{k=1}^{i} g_k^2}} g_i, \tag{3.20}$$

where g_k represents gradient at k^{th} update. η_0 is the initial learning rate. As we can see from this formula, the learning rate decreases gradually as the weights updates. Meanwhile, larger gradients yield smaller learning rate and smaller gradients generate larger learning rate on average. However, this method also relies on a manually selected global learning rate, and the learning rate will be infinitesimally small after a certain amount of iteration.

To deal with those two problems. Adadelta applies a per-dimensional basis learning rate. Instead of accumulating all the gradients, it accumulates over a certain time window. Nevertheless, storing previous gradients inefficient and require extra memory. Thus an exponentially decaying average of the past squared gradients is proposed to approximate a time window. In this case, updating rule for parameters $\vec{w}$ can be defined as

$$\Delta\vec{w}_i = -\frac{\eta}{\sqrt{\mathrm{E}[g^2]_i + \epsilon}} g_i, \tag{3.21}$$

where

$$\mathrm{E}[g^2]_i = \rho\mathrm{E}[g^2]_{i-1} + (1-\rho)g_i^2 \tag{3.22}$$

is the running average updated with regard to a constant ρ. ϵ is a constant value to better condition the denominator [31]. Note that denominator in Equation 3.21 is RMS of previous squared gradients, e.g.

$$\Delta\vec{w}_i = -\frac{\eta}{\mathrm{RMS}[g]_i} g_i. \tag{3.23}$$

If the parameter had some hypothetical units, the changes to the parameter should be changes in those units as well [31]. However, the authors noticed that the units in this update do not match. To solve this problem, another exponential

decay was proposed as

$$\mathrm{E}[\Delta\vec{w}^2]_i = \rho\mathrm{E}[\Delta\vec{w}^2]_{i-1} + (1-\rho)\Delta\vec{w}_i^2. \tag{3.24}$$

Therefore, root mean squared error of parameter updates will be

$$\mathrm{RMS}[\Delta\vec{w}]_i = \sqrt{\mathrm{E}[\Delta\vec{w}^2]_i + \epsilon} \tag{3.25}$$

Nevertheless, $\Delta\vec{w}$ for the current updating is not known, so an assumption is made that the curvature is locally smooth thus $\Delta\vec{w}$ can be approximated by computing the exponentially decaying RMS over a window of previous updates to give the updating rule as

$$\Delta\vec{w}_i = -\frac{\mathrm{RMS}[\Delta\vec{w}]_{i-1}}{RMS[g]_i} g_i. \tag{3.26}$$

3.3.3 Back-Propagation

All the optimization methods discussed above need the gradient in parameter space. We can easily calculate gradient for last layer as target outputs are given there. However, for any hidden layers in a DNN, there are no explicit target output, thus we cannot directly apply those optimization methods discussed above. Therefore here it comes Back-propagation (BP) for propagating errors backwards in a layer-wise manner through a neural network. It is commonly applied together with gradient-based methods discussed in Section 3.3.

Although ANNs showed their potential to model complex data, it was long a problem to train an ANN. The first efficient way widely used to implement optimization is back-propagation [26]. Here, we will use Multi-Layer Perceptrons [21] (MLPs) as an examples to illustrate Back-propagation (BP) algorithm.

Denote network input as $\vec{x} = [x_1, x_2, \cdots, x_n]$. Corresponding target output is denoted by $\vec{y} = [y_1, y_2, \cdots, y_m]$. Denote activation function for l^{th} layer as $\delta^{(l)}(\cdot)$, weight for i^{th} neuron in l^{th} layer connecting to its j^{th} input as w_{ji}, bias for l^{th} layer as b_l and output from i^{th} neuron in l^{th} layer as $y_i^{(l)}$. Back-propagation algorithm contains both feed-forward and back-propagation procedure. In feed-forward pass, activation values are calculated layer by layer till the final output. During back-propagation, the errors are back-propagated from top layer to the bottom one.

Let's consider first the feed-forward pass as it is much easier and intuitive. Denote w_{ji} as the weight connecting j^{th} input to i^{th} neuron. For the 1^{st} layer, as its neurons accept signal directly from input data, the weighted sum, $net_i^{(1)}$, can be calculated as

$$net_i^{(1)} = b_i^{(1)} + \sum_j w_{ji}^{(1)} x_j. \tag{3.27}$$

Activation values ${y_i}^{(1)}$ from this neuron will be

$$ {y_i}^{(1)} = \delta^{(1)}(net_i^{(1)}) = \delta^{(1)}(b_i^{(1)} + \sum_j w_{ji}^{(1)} x_j). \tag{3.28} $$

The weighted sum and activation value for i^{th} neuron in l^{th} layer will be

$$ {y_i}^{(l)} = b_i^{(l)} + \sum_j w_{ji}^{(l)} {y_j}^{(l-1)} \tag{3.29} $$

and

$$ {y_i}^{(l)} = \delta^{(l)}(net_i^{(l)}) = \delta^{(l)}(b_i^{(l)} + \sum_j w_{ji}^{(l)} {y_j}^{(l-1)}) \tag{3.30} $$

respectively. Therefore, at the output layer (denoted as L^{th} layer), the final activation will be

$$ {y_i}^{(L)} = \delta^{(L)}(b_i^{(L)} + \sum_j w_{ji}^{(l)} {y_j}^{(L-1)}). \tag{3.31} $$

Here, we use the notation C to represent network cost to minimize, e.g.

$$ C = f(\vec{y},\ \vec{y}^{(L)}), \tag{3.32} $$

where $\vec{y}^{(L)}$ represents output values from L^{th} layer, e.g. $\vec{y}^{(L)} = [{y_1}^{(L)}, {y_2}^{(L)}, \cdots, {y_m}^{(L)}]$. $f(\cdot)$ is the function maps two vectors into an error metric. For instance, it could be mean squared error:

$$ f(\vec{y}, \vec{y}^{(L)}) = \frac{1}{m} \sum_i^m ({y_i}^{(L)} - y_i)^2. \tag{3.33} $$

When we have reached the output layer, the error can be back-propagated through the whole network to the first layer by applying chain rule. For output layer, its error, $e_i^{(L)}$ can be calculated by

$$ \begin{aligned} e_i^{(L)} &= \frac{\partial C}{\partial net_i^{(L)}} \\ &= \frac{\partial C}{\partial y_i^{(L)}} \frac{\partial y_i^{(L)}}{\partial net_i^{(L)}} \\ &= \frac{\partial C}{\partial y_i^{(L)}} [\delta^{(L)}]'(net_i^{(L)}). \end{aligned} \tag{3.34} $$

For l^{th} layer except for the output layer, its errors can be defined as

$$ e_i^{(l)} = \frac{\partial C}{\partial net_i^{(l)}} = \sum_k \frac{\partial C}{\partial net_k^{(l+1)}} \frac{\partial net_k^{(l+1)}}{\partial net_k^{(l)}}. \tag{3.35} $$

In Equation 3.35, $\frac{\partial C}{\partial net_k^{(l+1)}}$ is actually $e_k^{(l+1)}$ and

$$\frac{\partial net_k^{(l+1)}}{\partial net_k^{(l)}} = \sum_i w_{ik}^{(l+1)} [\delta^{(l)}]'(net_k^{(l)}). \tag{3.36}$$

Substituting this into Equation 3.35 and finally we will have gradient for each weight as

$$\begin{aligned} \frac{\partial C}{\partial w_{ji}^{(l)}} &= \frac{\partial C}{\partial net_i^{(l)}} \frac{\partial net_i^{(l)}}{\partial w_{ji}^{(l)}} \\ &= \sum_k w_{ji}^{(l+1)} e_k^{(l+1)} [\delta^{(l)}]'(net_k^{(l)}) y_k^{(l-1)} \\ &= y_k^{(l-1)} e_i^{(l)} \end{aligned} \tag{3.37}$$

3.4 Initialization

Performance of a neural network is heavily affected by its initial state as it usually applies one of the non-convex optimization methods discussed in previous section. When there are more than one local minimum, a non-convex solution might converge to different optimal points for different initial conditions. The problem here is, not all the local minima give similar solutions. There are always some local optimal points generalizing much worse than others. Therefore, initialization method applied in an ANN will have an important impact on the network performance.

Xavier Initialization, or Glorot Initialization, is an initialization strategy proposed by Xavier Glorot and Yoshua Bengio in 2010 [6]. It was proposed to deal with gradient saturation problems and achieved great success to improve network performance.

Consider a neuron which takes n inputs $\vec{x} = [x_1, x_2, \ldots, x_n] \in \mathbb{R}^n$ and gives a single output $\hat{y} \in \mathbb{R}$. Its weights are denoted by $\vec{w}$. As discussed in Section 2.1, the output can be calculated as

$$\hat{y} = w_1 x_1 + w_2 x_2 + \cdots + w_n x_n + b, \tag{3.38}$$

where b is the bias. As the weights $\vec{w}$ and input $\vec{x}$ are independent, each item $w_i x_i$ will have variance value (denoted as $\mathrm{Var}(w_i x_i)$) which can be calculated by

$$\mathrm{Var}(w_i x_i) = \mathrm{E}[x_i]^2 \mathrm{Var}(w_i) + \mathrm{E}[w_i]^2 \mathrm{Var}(x_i) + \mathrm{Var}(x_i)\mathrm{Var}(w_i), \tag{3.39}$$

where $\mathrm{E}[x_i]$ represents the expectation of x_i. Assume that the inputs and weights are normalized so that they have zero-mean, the variance will become

$$\mathrm{Var}(w_i x_i) = \mathrm{Var}(x_i)\mathrm{Var}(w_i). \tag{3.40}$$

Here, if a further assumption is made that x_i and w_i both are independent and identically distributed (i.i.d.), we will have

$$\begin{aligned} \mathrm{Var}(\hat{y}) &= \mathrm{Var}(w_1x_1 + w_2x_2 + \cdots + w_nx_n + b) \\ &= n\mathrm{Var}(x_i)\mathrm{Var}(w_i). \end{aligned} \tag{3.41}$$

To be in conformity with previous sections, we use $y^{(l)}$ to represent output value from l^{th} layer in a network, $n^{(l)}$ for data dimensionality fed to l^{th} layer and $w^{(l)}$ for its weights. From discussions above, variance of output value from a network which has L layers will be

$$\begin{aligned} \mathrm{Var}(y^{(L)}) &= n^{(L)}\mathrm{Var}(y^{(L-1)})\mathrm{Var}(w^{(L)}) \\ &= n^{(L)}[n^{(L-1)}\mathrm{Var}(y^{(L-2)})\mathrm{Var}(w^{(L-1)})]\mathrm{Var}(w^{(L)}) \\ &= n^{(L)}n^{(L-1)}[n^{(L-2)}\mathrm{Var}(y^{(L-3)})\mathrm{Var}(w^{(L-2)})]\mathrm{Var}(w^{(L-1)})\mathrm{Var}(w^{(L)}) \\ &\vdots \\ &= \mathrm{Var}(x)\prod_{l=1}^{L}[n^{(l)}\mathrm{Var}(w^{(l)})]. \end{aligned} \tag{3.42}$$

The product item in Equation 3.42 is problematic for DNNs. If every single item in this product is greater than 1, it will reach a significantly large value at the end, preventing networks from learning interesting representations. On the other hand, the top layers will only be able to get and generate near-fixed values, meaning a network will never leave its initial state. Therefore, the best choice would be keeping data variance unchanged from input to output. For this purpose, it is needed to keep

$$n^{(l)}\mathrm{Var}(w^{(l)}) = 1. \tag{3.43}$$

Consider back-propagation procedure, errors propagated from output layer will finally vanish if gradients propagated by each layer has smaller variance than that is received. Or they explode when the variance gets greater for each layer. Similarly, we will need

$$n^{(l+1)}\mathrm{Var}(w^{(l)}) = 1 \tag{3.44}$$

to keep the gradients propagated in a suitable range. In Xavier's method, a harmonic average is applied to take both forward and backward procedure into consideration. That is, for l^{th} layer it a DNN, variance of its weights, denoted by $\mathrm{Var}(w^{(l)})$, should be initialized as

$$\mathrm{Var}(w^{(l)}) = \frac{2}{n^{(l)} + n^{(l+1)}} \quad \text{with } l = 1, 2, \cdots, L-1. \tag{3.45}$$

As a result, for normal and uniform distribution, the weights $w^{(l)}$ should be initialized from

$$w^{(l)} \sim N(0,\ \frac{2}{n^{(l)} + n^{(l+1)}}) \tag{3.46}$$

or

$$w^{(l)} \sim U[-\frac{\sqrt{6}}{\sqrt{n^{(l)} + n^{(l+1)}}},\ \frac{\sqrt{6}}{\sqrt{n^{(l)} + n^{(l+1)}}}] \tag{3.47}$$

respectively.

Although it may seem to be too much assumptions, Xavier Initialization has been applied in many experiments and it is a popular option in many deep learning frameworks [66; 67; 70; 71].

There are some other similar strategies for initializing a DNN. For instance, LeCun et al. [72] used $\sqrt{\frac{3}{n^{(l)}}}$ as the scale factor for uniform distribution to initialize $l^{(th)}$ layer in a neural network. He et al. [15] applied $\sqrt{\frac{2}{n^{(l)}}}$ as the variance for normal distribution.

3.5 Unsupervised pre-training with autoencoders

As discussed in Section 2.3, an autoencoder tries to reconstruct its inputs at output end, which guarantees that signals passing through the network will carry the information needed to reconstruct original data. If there are some constraints preventing autoencoders from copying input data to output (for example, reduce dimensionality somewhere between input and output), they will have to learn the intrinsic representations of given data with a lower dimensionality. This is a very interesting property which had made unsupervised pre-training with autoencoders gain its popularity around one decade ago.

There are two common ways to apply autoencoders for unsupervised pre-training. One is applying a single autoencoder to training data. After the model converges, copy the encoder part out and stack one or several layer on top of bottleneck layer to form a classifier. The other method applies stacked autoencoders [5] to pre-train a network.

A stacked autoencoder typically applies symmetrical architecture so that is can be pre-trained in a greedy layer-wise order. Take the stacked autoencoder shown in Figure 3.3 as an example, it can be seen as a stack of two individual autoencoders. When training this network, it will first train an autoencoder with input/output dimensionality and bottleneck dimensionality 4, e.g. the "Hidden" layer shown in the figure. After the model converges, the representation used in its bottleneck will be use as input and target output to train another autoencoder. After this stage, the two autoencoders can be stacked together as shown in Figure 3.3 and fine-tuned with original data. This procedure can be repeated as many times as wanted to stack

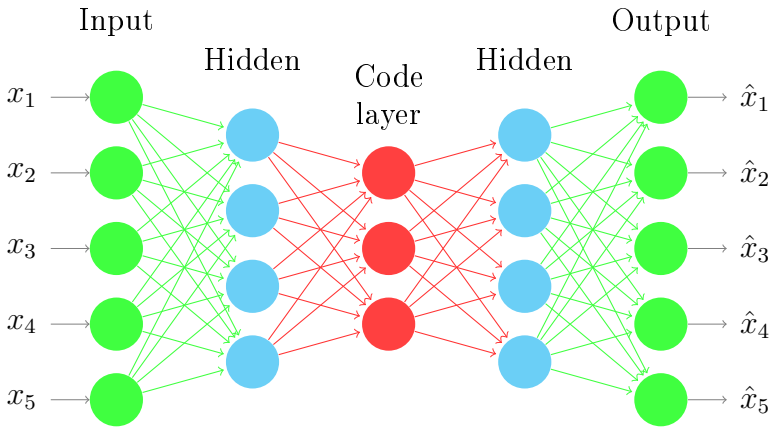

Figure 3.3: Example of a stacked autoencoder.

new layers on top of existed ones. By taking the encoder out from fine-tuned stacked autoencoder and adding new layers to the bottleneck layer, a new network will be built with pre-trained layers from the encoder. With this pre-training strategy, pre-trained network proposed by Cireşan et al. [42] created state-of-art results for digit recognition and object classification in 2011.

Restricted Boltzmann Machines (RBMs) [3] are another popular choice for greedy layer-wise pre-training. They learn a probability distribution over input samples. In a work by Bengio et al. [2], theoretical and experimental comparison was made for RBMs and autoencoders. It was shown that autoencoders can be used like RBMs for layer-wise pre-training and achieve similar performance. An experiment [42] shows that denoising stacked autoencoders can initialize a network even better than the RBMs. Besides comparison between RBMs and autoencoders, there was also a work trying to combine them together to see whether we can benefit from this combination. This work given by Tan et al. [73] shows that combining them together works better than using a stacked autoencoder only.

As discussed above, building encoder in a decreasing order with regard to its layer dimensionality is a common way to regularize an autoencoder. However, it is indicated in a work [2] that even the autoencoder is not built with a decreasing order, it might still generalize well.

Paine et al. [9] studied how unsupervised pre-training performs for different dataset size and found that unsupervised pre-training helps when there are much more unlabeled data available than labeled ones but it hurts when labeled samples have larger quantity than unlabeled ones or there is only labeled data available.

3.6 New ways applying autoencoders for pre-training

Two pre-training strategies are proposed in this work, one of which is a supervised variant. Instead of reconstructing individual samples as well as possible, it tries to reconstruct same target for data from same class in order to erase intra-class variance. The other one adds the output layer of target network (e.g. softmax layer for a classifier) as bottleneck layer of an autoencoder.

3.6.1 Embedding target network as the encoder part for unsupervised pre-training

It can be seen from Section 3.5 that conventional pre-training methods with autoencoders normally initialize the bottom layers only for a classifier. Newly-added layers (a single softmax layer in some cases) are still randomly initialized. Here, a new method is proposed with the whole classifier embedded as the encoder part. After the autoencoder converges, all the layers of the classifier are initialized with pre-training.

The output layer in a classifier typically has the smallest dimensionality. Therefore, it will be definitely more information loss if we have only few neurons with softmax activation at bottleneck layer. However, there are two potential advantages is has to form an autoencoder in this way. First, it reduces model complexity by reducing number of weights for bottleneck layer and its successor. Consider a classifier for 10-class classification task. If the layer connected to output has 200 neurons and input data which is desired to be reconstructed directly from those neurons has the dimensionality 784, it will need $200 \times 784 = 156.8K$ wights (excluding bias). On the other hand, if one more softmax layer is added in between with 10 nodes, total weights (excluding bias) needed will be reduced to $200 \times 10 + 10 \times 784 = 9.84K$, which is 93.72% fewer then the first case. The second advantage is that the whole classifier will be initialed by pre-training, including the final output layer.

In summary, given a target classifier, several new layers will be stacked on top of its output layer to form an autoencoder. After the autoencoder converges, its encoder part can be taken out for fine-tuning with labeled data.

3.6.2 Supervised variant of autoencoder

As discussed in Section 3.5, an autoencoder tries to construct every single input as its output, which means it preserves all the detailed information needed to reconstruct the data. In that case, intra-class variance is also preserved to distinguish the samples from each class. However, for classification tasks, we barely care about intra-class variance. No matter how they vary from each other in the same category,

their labels are always the same. That is, discriminative tasks does not need as much information as that for generative tasks. Therefore, a supervised variant of autoencoder is proposed for discriminative tasks like classification.

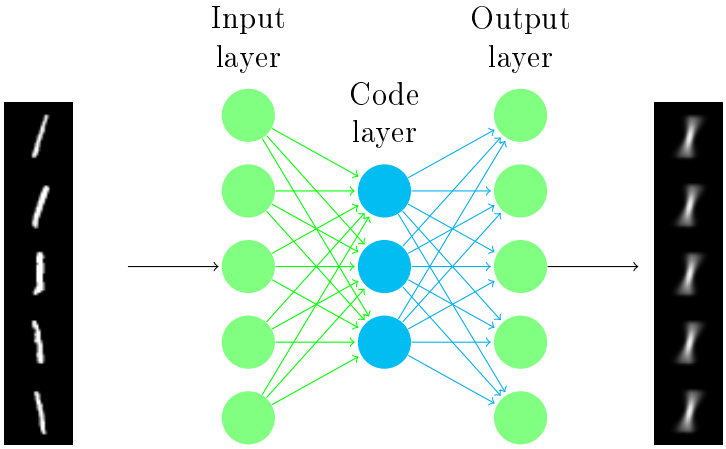

Figure 3.4: Supervised variant of autoencoder. No matter how those "1" differ from each other, their target outputs stay the same.

Instead of reconstructing every individual input itself, the proposed method tries to reconstruct *same* target output for samples from the same class. As a result, it will intuitively erase the intra-class variance and keep inter-class difference at the same time. Meanwhile, same strategy discussed in previous method is applied for building the variant of autoencoder, e.g. the whole classifier is embedded as encoder part. Mean Squared Error is used as objective function to minimize when training the supervised autoencoder while softmax is the activation for a multi-class classifier. Therefore, this variant is similar to a special kind neural networks having more than one loss function proposed by Xu et al. [74].

In short, given a target network, several new layers will be stacked on top of its output layer to form a supervised variant of autoencoder. Target output of this variant is always the same for samples from same category but different from each other. After the new network converges, the encoder part, which contains target network layers, can be taken out for fine-tuning.

4. EXPERIMENTS

This chapter introduces datasets used in the experiments, explains evaluation in detail and presents the results.

4.1 Datasets

To guarantee an easy and simple comparison, experiments are conducted on three widely used benchmarks: MNIST [75], CIFAR10 and CIFAR100 [76].

4.1.1 MNIST

MNIST [75] is abbreviation for Mixed National Institute of Standards and Technology. It is a database of handwritten digits, containing 60000 training samples and 10000 test images in gray scale format from 10 categories representing digit 0 to 9. All the images from this dataset are centered and normalized to fixed size 28x28. Image samples are shown in Figure 4.1.

Figure 4.1: Sample images taken from MNIST dataset.

Digits in MNIST are not strictly evenly distributed. As shown in Figure 4.2, amount of samples varies for different categories. Especially, there are 1564 more images for digit "1" than "5". However, it is still roughly uniformly distributed considering the fact that there are 70000 images in total and the intra-class variance is also small. MNIST has been widely used during past two decades and many state-of-art results [29; 77; 78; 79] have been reported on this dataset with "near-human performance", e.g. over 99 %.

4.1.2 CIFAR10

CIFAR10 is a widely applied benchmark for object recognition and classification. It is a subset of larger *80 Million Tiny Images* [80] dataset. Similar to MNIST, it also contains images from ten categories. The different thing is, those ten categories are

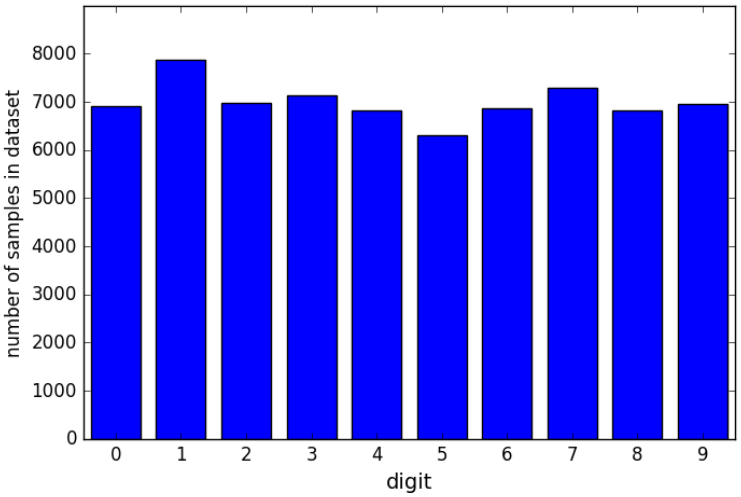

Figure 4.2: Distribution of different digits.

strictly balanced, meaning there are same amount images for each class. Categories include animals and vehicles: bird, cat, dog, deer, frog, horse, ship, truck, airplane and automobile. There are 60000 samples in total, 50000 of which are conventionally used for training and remaining images comprise a test set. All the images are in RGB format and are fixed to size 32x32. Samples taken from this dataset can be seen in Figure 4.3.

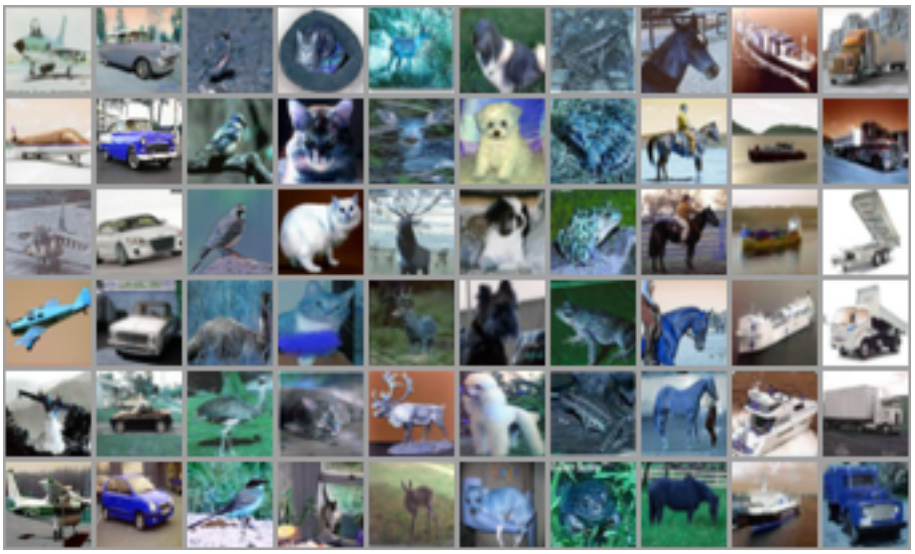

Figure 4.3: Example images taken from CIFAR10 dataset, with each column representing a category.

CIFAR10 is more challenging to classify compared with MNIST as there is much higher intra-class variance because of different color, angle and complicated background.

4.1.3 CIFAR100

CIFAR100 [76] has same amount of images, image format and size as CIFAR10. The significant difference is that there are 100 categories in this dataset thus much less samples for each category, making it markedly challenging than CIFAR10. However, CIFAR100 dataset can also be used for an easier twenty-class classification task as it divides all its 100 categories into 20 super classes, each containing 5 base categories. That is, each image in CIFAR100 has two labels, a "fine" label that describes what is in this image and a "coarse" label telling to which super class this image belongs. For instance, if an image has a "fine" label "beaver", it will always have a "coarse" label "aquatic mammals".

4.1.4 Data Augmentation

When training a deep network on a small dataset containing complex objects, the network can easily get over-fitted [81]. In that case, a network performing considerably well on training set will give poor results on test set, meaning it cannot generalize well. As a special kind of regularization, applying data augmentation on *training* set can prevent networks from over-fitting thus promise better performance. As a result, a huge amount of experiments conducted on CIFAR10 and CIFAR100 applied data augmentation techniques. We follow this idea and apply data augmentation on CIFAR10 dataset. Although there are many different ways to deform input data to obtain more training samples, most popular techniques applied to images are those below or a combination of them.

Horizontal Flip means flipping an image along its width (horizontal axis).
Vertical Flip means flipping an image along its height (vertical axis).
Width Shift means shifting an image along its horizontal axis.
Height Shift means moving an image along its vertical axis.
Channel Shift shuffles image channels randomly.
Zooming scales images so that it looks larger or smaller.
Rotation rotates the image with a given angle
Shearing shears image intensity with a certain angle.

Figure 4.4 gives a basic idea how those techniques deform the original image in different ways. In this thesis work, horizontal flip, height and width shift, zooming and rotation are applied for data augmentation. As MNIST is a simple dataset with enough amount of training samples, data augmentation techniques are applied to CIFAR10 only.

Figure 4.4: A simple illustration on data augmentation techniques used in this thesis. Original image locates in the center. The image in top left corner is generated by rotating original image with 12°; the image below it is generated by vertical shifting; the left bottom one is generated by shearing intensity by 0.5 radians; bottom center one is generated by horizontal shifting and the top center one is generated by shuffle the color channels; the top right one and the one below it are generated by flipping the image along horizontal and vertical axis respectively. By zooming the images to a ratio 115%, we can get the right bottom one.

4.2 Evaluation

This section gives an description on how the networks are built, trained and evaluated.

4.2.1 Overview

Experiments are conducted on MNIST, CIFAR10 and CIFAR100 datasets as discussed in previous section. All the classifiers are optimized using SGD with learning rate to be 0.01. Learning rate decays 1.0×10^{-5} for each updating and momentum is applied with value 0.9.

For MNIST dataset, comparison is made among a classifier initialized from scratch, a classifier initialized with a conventional autoencoder as discussed in Section 3.5 and two classifiers initialized in the way discussed in Section 3.6. Especially, target images for supervised variant of autoencoder are generated by averaging all the samples from same category in MNIST dataset. As we can see in Figure 4.5, those images are highly recognizable to human beings.

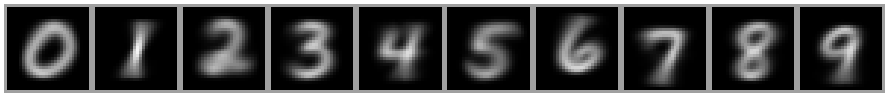

Figure 4.5: Images generated by averaging all training samples from each category in MNIST dataset.

While SGD is a good choice for testing different initialization methods of a network because of its sensitivity to initial state, it is problematic for training a deep network. Therefore, Adadelta is used as optimization method for training autoencoders. As a result, another classifier which is initialized from scratch but optimized with Adadelta is trained to see whether performance improvement comes from a better initialization or a different optimization mechanism.

For CIFAR10 dataset, it does not make much sense to average the input images from same category due to extremely high intra-class variance. As shown in Figure 4.6, the average images can barely be recognized by a human being. Therefore, the target images used for supervised variant of autoencoder in this case are also those average images taken from MNIST. As we will see later in this section, using those unrelated images can still improve network performance significantly.

Figure 4.6: Images generated by averaging all training samples from same category in CIFAR10 dataset

As those experiments mainly concern classification tasks, classification accuracy is the metric used to measure network performance. However, as a supplement, cross-entropy for each case, along with some other statics, is also presented in Appendix B.

4.2.2 Software

All the experiments are implemented with a deep learning framework called Keras [71]. It is a python library capable of running on top of a back-end, either Theano [66] or TensorFlow [67]. As both back-ends used by Keras are able to take advantage of computational power provided by GPUs, it can also benefit from using high speed GPUs. It aims at fast experimentation and is thus very simple for prototyping. By 25^{th} September, 2016, Keras has 265 contributors on github, making it 2^{nd} place among TensorFlow, Theano, Caffe[70] and other popular deep learning frameworks (See Appendix D for statistics). Appendix A gives an example on how to use Keras to build, train and test a neural network.

4.3 Results

This section shows experiment results conducted on MNIST, CIFAR10 and CIFAR100 with different network architecture.

4.3.1 Networks trained on MNIST

There are three different network architectures tested on MNIST, each divided into five different cases: a normal classifier initialized from scratch with SGD optimization, a normal classifier initialized from scratch and optimized with Adadelta, a classifier initialized with a conventional autoencoder as discussed in Section 3.5, a classifier initialized by embedding whole classifier as the encoder part of an autoencoder, and a classifier initialized by embedding it as the encoder part of a supervised variant of an autoencoder as described in Section 3.6.

First network trained on MNIST

The first network tested on MNIST is very simple, 64 feature maps are generated by applying 64 convolutional kernels with size 3×3 on input images and then pooled by a 3×3 max-pooling operation with strides 2×2. The feature maps are then flattened to a long vector and fully-connected to a layer with 300 nodes. Activation function for those layers is ReLU. Another fully-connected layer is stacked on top of this 300-node layer with softmax activation as final output. All the networks' topology can be found in Appendix C.

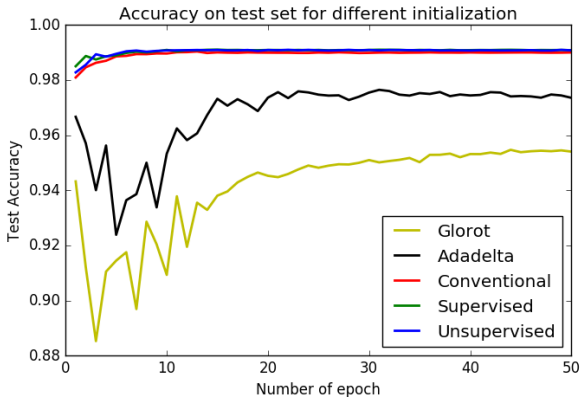

Figure 4.7: Classification accuracy on test set. "Glorot" means a classifier initialized with Glorot Initialization; "Adadelta" shows classifier initialized with same initialization but optimized with Adadelta; "Conventional" means a classifier initialized with a conventional autoencoder; "Supervised" and "Unsupervised" indicate proposed method, among which "supervised" means the supervised variant.

As shown in Figure 4.7, all the pre-trained classifiers outperform the randomly

initialized ones. For those two classifiers initialized with Glorot Initialization, classifier optimized with Adadelta gives better performance than the one using SGD. Both supervised variant and unsupervised autoencoder give 0.1% higher accuracy than the classifier initialized with a conventional autoencoder.

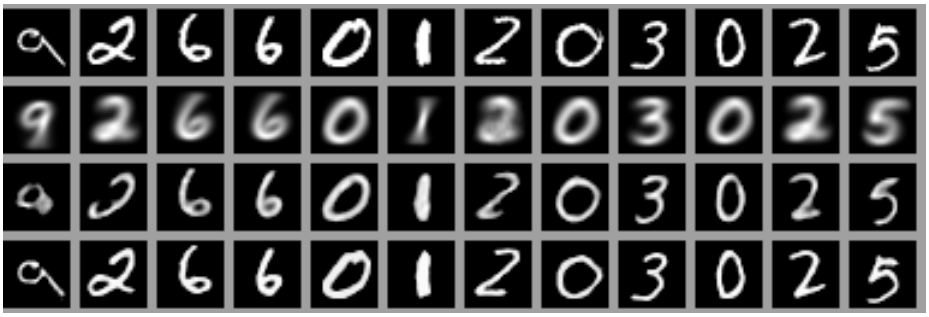

Figure 4.8: Images reconstructed by autoencoders. Top row presents original image. The second row depicts how well a supervised variant of autoencoder can reconstruct its input. Images reconstructed by the autoencoder whose bottleneck layer is comprised by only 10 softmax activated neurons are shown below it. Samples shown in the bottom row are generated by a conventional autoencoder with bottleneck layer dimensionality 300.

Figure 4.8 shows the reconstruct results for autoencoders. Original images are shown in the top row and reconstructed results are listed in the bottom. As shown in the figure, the conventional autoencoder gives fairly good result that the reconstruction error is trivial. Interesting thing here is, there are only 10 neurons for the autoencoders used to reconstruct images shown in the second and third rows, which the autoencoders can reconstruct input data considerably good with a compression rato 78.4:1.

Second network trained on MNIST

It has been long realized [82; 83; 12] that a deep network is able to give compatible or better performance than shallow ones with even less parameters. Based on this idea, the second network is deeper but with fewer parameters. The input images are convoluted with sixteen 3×3 kernels, followed by another convolution layer with forth-eight 3×3 kernels, both followed by the same pooling strategy applied to the first network above. The fully connected layers have same number of nodes with those layers in the first network. Therefore, the second network has one more convolutional layer and one more pooling layer than the first one with 92.6% fewer parameters in total.

As it shows in Figure 4.9, classifier optimized with Adadelta performs slightly worse than its counterpart optimized with SGD. Pre-trained classifiers still outperform the randomly initialized ones where the supervised variant gives trivially poorer

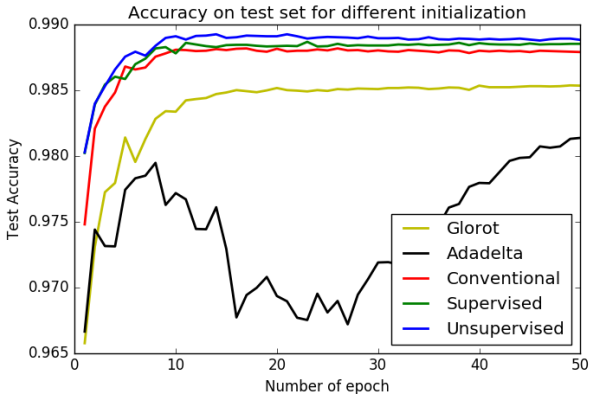

Figure 4.9: Classification accuracy on test set

results than the unsupervised autoencoder, yet still show a better performance than the conventional autoencoder.

Third network trained on MNIST

One more convolutional layer containing forty-eight 3×3 kernels is added to the second network, together with an additional max-pooling layer. Those newly-added layers are inserted between the last pooling layer and first fully-connected layer in the second network discussed in previous paragraphs.

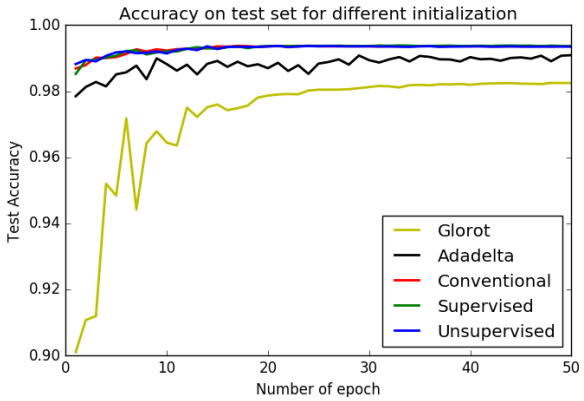

Figure 4.10: Classification accuracy on test set

From Figure 4.10, it shows same trend shown in first network. That is, pre-trained classifiers yield better performance than the randomly initialized ones, where network optimized with Adadelta performs slightly better. Difference among pre-trained classifiers is trivial.

4.3.2 Networks trained on CIFAR10

Similar to MNIST cases, there are three different networks tested on CIFAR10. For each network, there are five classifiers initialized in different ways same as those trained on MNIST. Nevertheless, CIFAR10 is considerably more challenging than MNIST, thus classifiers trained on it are deeper.

Data augmentation techniques are applied on CIFAR10. To make clear how well data augmentation techniques help with network performance, five more classifiers initialized the same way as those described above are trained on the augmented dataset. Therefore, for each network topology, there will be ten classifiers in total, five of which are trained on original dataset while the others are optimized on the deformed images. To guarantee an intuitive comparison, the autoencoders are always trained with original samples only. Augmented dataset is used for optimizing classifiers.

First network trained on CIFAR10

Similar to MNIST cases, we start from a shallow classifier. Sixteen 3×3 convolution kernels are applied to extract feature maps from input image, followed by a 4×4 max-pooling with stride 2 along both axes. All the feature maps are then flattened to a long vector and connected to a fully-connected layer with 300 nodes, which is followed by a softmax output layer with ten nodes.

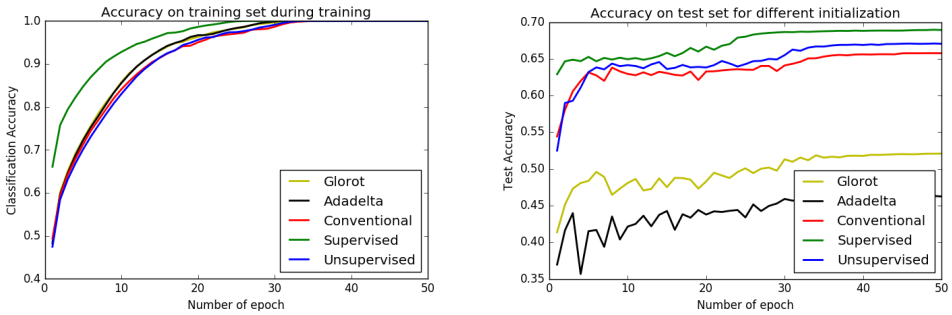

Figure 4.11: Classification accuracy on training and test set without data augmentation.

As shown in Figure 4.11, pre-trained networks give much better performance when no data augmentation techniques are applied. Especially, classification accuracy of pre-trained classifiers at the end of their first epoch is already higher than that of randomly initialized classifiers when they are at the end of 50^{th} epoch, where the network performance has stabilized.

In terms of training, all five classifiers converge within 40 epochs. The classifier initialized with supervised variant of autoencoder converges fastest among all classifiers.

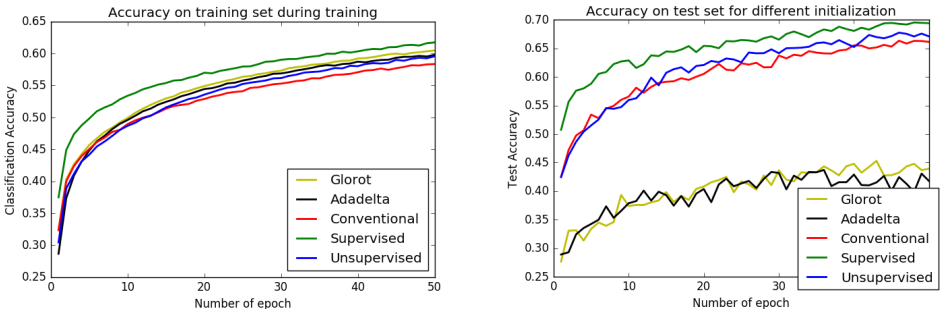

Figure 4.12: Classification accuracy on training and test set with data augmentation.

When data augmentation techniques are applied, the classifiers converge slower. As we can see from the left part of Figure 4.12, training accuracy still improve gradually after 40 epochs. From right part, pre-trained classifiers still yield better performance than the randomly initialized ones. Especially, classifiers trained from scratch yield similar performance, regardless they are optimized with SGD or Adadelta. The one pre-trained by a supervised variant of autoencoder outperforms all other classifiers.

If we take a look at reconstructed images by autoencoders shown in Figure 4.13, the supervised variant can still reconstruct the digits accordingly. For the other autoencoder where the bottleneck dimensionality is just 10, the reconstructed images are not recognizable to human beings. For the autoencoder with 300 nodes as its bottleneck layer, the reconstructed images are similar to original ones although they are blurred. The interesting thing here is, even those three autoencoders give quite different reconstruction as discussed above, classifiers initialized by them perform more or less similar as shown in Figure 4.11.

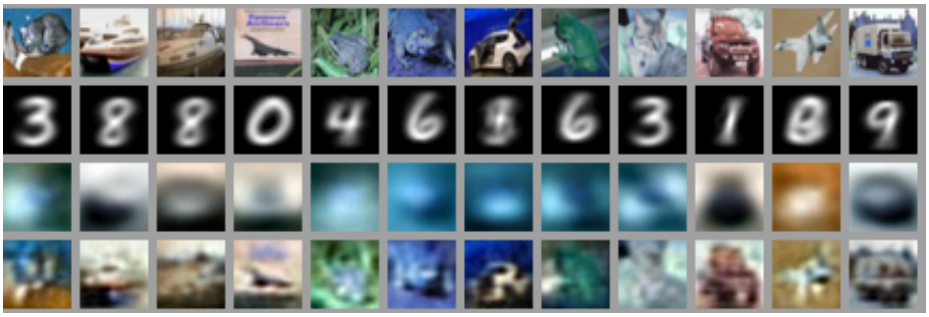

Figure 4.13: Images reconstructed by autoencoders. Top row presents original image. The second row depicts how well a supervised variant of autoencoder can reconstruct its input. Images reconstructed by the autoencoder whose bottleneck layer is comprised by only 10 softmax activated neurons are shown below it. Samples shown in the bottom row are generated by a conventional autoencoder with bottleneck layer dimensionality 300.

Second network trained on CIFAR10

For the second network, a convolutional layer with sixteen 3×3 kernels is used to accept input data. The feature maps are pooled with a 2×2 region and then convolved with forty-eight 3×3 kernels. Those new feature maps are followed by a max-pooling layer with 4×4 overlapped windows where the strides along both axes is 2. After pooling operation, the feature maps are flatten into a long vector and two more fully-connected layers are stacked on top of that as every classifier is in previous sections.

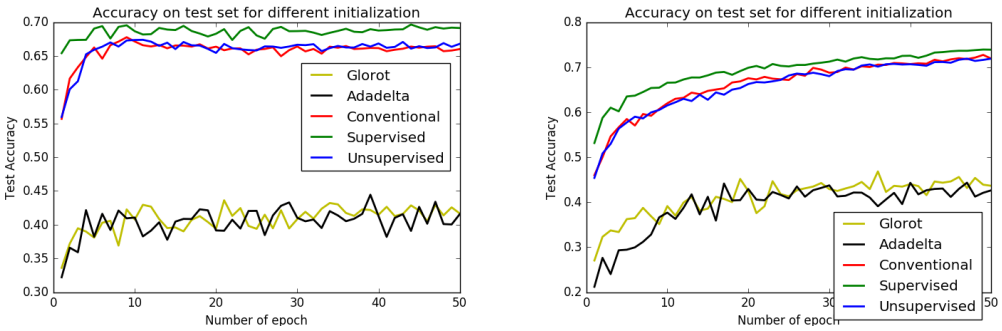

Figure 4.14: Classification accuracy on test set, with (right part) and without (left part) data augmentation.

As shown in Figure 4.14, pre-trained classifiers still perform much better than the randomly initialized ones, no matter they are optimized with Adadelta or SGD. Especially, supervised variant still gives best performance among all the networks. Data augmentation techniques help with improving network performance.

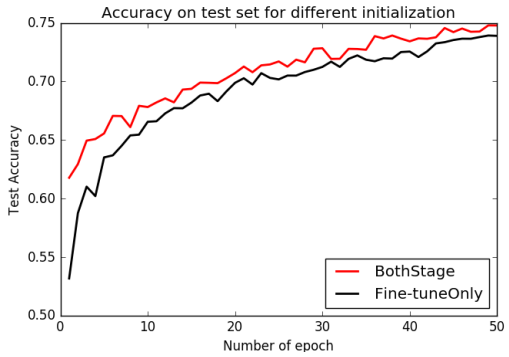

Figure 4.15: Classification accuracy on test set with data augmentation applied to only fine-tuning (black) or applied to both pre-training and fine-tuning (red)

Data augmentation is applied only when training the classifier, it is not used for pre-training stage. To see whether it also help pre-training, more experiments are conducted with data augmentation applied to both pre-training and fine-tuning. Results shown in Figure 4.15 indicates that it helps improving pre-training as well.

4.3.2.1 Third network trained on CIFAR10

As mentioned above, CIFAR10 is more difficult than MNIST, which might need more representative models. Therefore, the third network is deeper than all its successors. The deepest classifier trained for MNIST has only two convolutional layers and two fully-connected layers (the layer flattening feature maps into a long vector is not counted). However, there are four convolutional layers and three fully-connected layers. In a layer-sequential order, they are a convolutional layer with 32 kernels, a max-pooling layer extracting maximum values from 3×3 regions with stride 2 along both axes, two convolutional layers with 64 and 72 kernels respectively, a max-pooling layer with same regime as the previous one, one more convolutional layer with 148 kernels. All the convolutional kernels are of size 3×3 and overlapped with its neighbors. Those 148 maps are flattened into a long vector and fully-connected to a layer with 800 neurons, followed by a 300-node layer and a 10-node softmax layer as output. ReLU is applied as activation function for all the parametric layers except for the output layer.

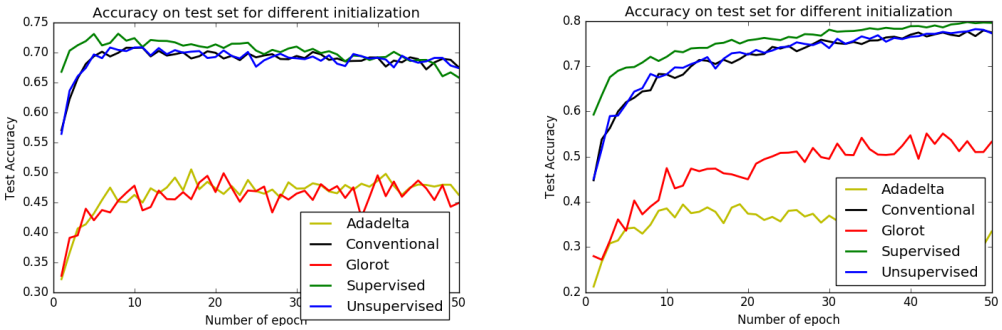

Figure 4.16: Classification accuracy on test set, with (right part) and without (left part) data augmentation.

From Figure 4.16, the pre-trained classifiers still perform much better than those trained from scratch and supervised variant of autoencoder shows best performance again. However, as can be seen in the left part, the pre-trained classifiers begin to over-fit after around 10 epochs. With data augmentation, the networks keeps improving except for the one optimized with Adadelta. This Meanwhile, data augmentation improves classification accuracy, which can be drawn by comparing the two images in Figure 4.16. Applying data augmentation techniques to also pre-training stage brings further improvement, just similar to the case in Figure 4.15.

4.3.3 Networks trained on CIFAR100

Just for checking whether pre-training helps with larger datasets (e.g. more categories), CIFAR100 is used to compare pre-trained classifiers and those trained from scratch. However, taking average images for CIFAR100 does not make much sense, yet it is not an easy job to find 100 different simple images which can be used as target output for the supervised variant of autoencoder. Therefore, the supervised variant is not checked in this experiment.

There are two different network architecture applied in the experiment, which can be found in Appendix C. The classification on test set for both networks are shown in Figure 4.17

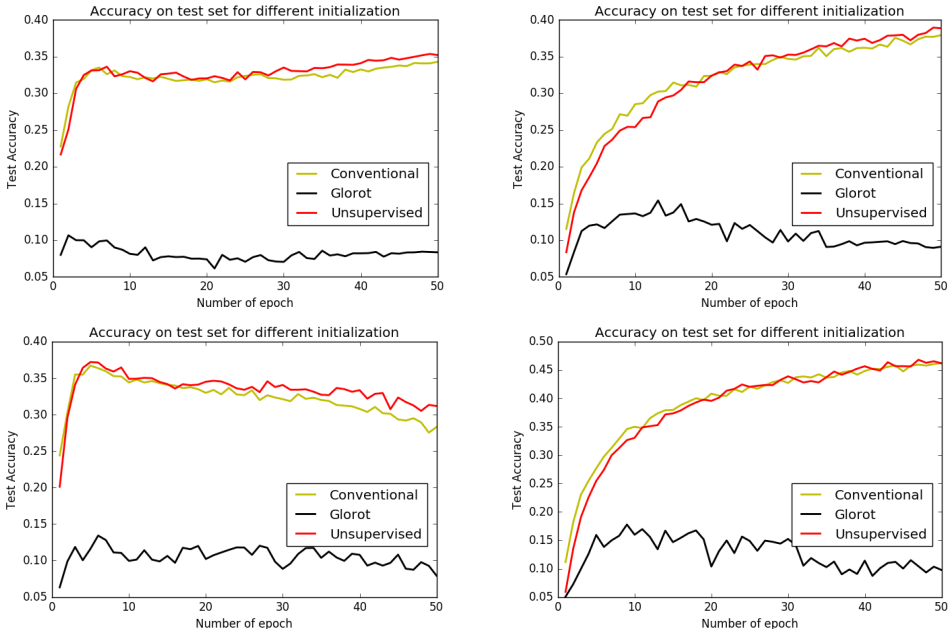

Figure 4.17: Classification accuracy on test set for the networks. Each row represents a different network topology, without (left) and with (right) data augmentation.

As we can see, pre-trained networks show better performance than those initialized randomly and trained from scratch. The network whose test accuracy is shown in second row is deeper than the other one and shows over-fitting when no data augmentation techniques are applied. In both networks, pre-training initializes classifier to a better basin in parameter space, which improves performance for SGD optimization.

5. CONCLUSIONS

In this thesis work, we studied initializing a classifier with pre-trained autoencoders. Two new ways of using autoencoders or their variants to pre-train a classifier have been proposed to improve network performance.

Experiments have been conducted on three computer vision benchmarks, including MNIST, CIFAR10 and CIFAR100. MNIST is a dataset containing huge number of handwritten digits. As it is a relatively easy task, modern neural networks like CNNs can achieve classification accuracy higher than 98% with very simple architecture. Using proposed pre-training strategy further improved network performance from 96% to 98.2% for a shallow network and from 98.36% to 99.35% for a deeper one. For more difficult tasks like CIFAR10 and CIFAR 100, it is difficult for autoencoders to reconstruct input images perfectly. However, they were still able to extract intrinsic information from data and initialize classifiers to a better starting point. We saw improvements from 56.65% to 79.8% for CIFAR10 and from 15.2% to 47.5% for CIFAR100 with data augmentation techniques.

From the experiments, pre-training helps with network performance concerning classification tasks. Using a softmax layer as the bottleneck can give similar performance as applying a conventional autoencoder where the bottleneck dimensionality is much higher. Furthermore, the supervised variant of the autoencoder shows slight better performance than the conventional autoencoders,

Although this proposed variant of autoencoder showed its superiority over conventional autoencoders in terms of pre-training, it needs labeled data. For future work, there might be other possibilities to pre-train a neural network in completely unsupervised manner. Also, those experiments concerned only simple networks. Yet there are many complicated networks which have shown impressive performance like VGGNet [33], ResNet [34], Multi-Column DNN [78] and so forth. It is not clear at this moment whether those pre-training methods will further improve performance for those networks.

REFERENCES

[1] Arthur L Samuel: "Some studies in machine learning using the game of checkers". *BM Journal of research and development*, vol. 3(3), p210–229, 1959

[2] Yoshua Bengio, P. Lamblin, D. Popovici, H. Larochelle: "Greedy layer-wise training of deep networks". *Advances in neural information processing systems*, vol. 19, p153-160, 2007

[3] Geoffrey E Hinton, S. Osindero, Y. Tech: "A fast learning algorithm for deep belief nets". *Neural Computation*, vol. 18(7), p1527-1554, 2006

[4] Geoffrey E Hinton, R. R. Salakhutdinov: "Reducing the dimensionality of data with neural networks". *Science*, vol. 313, p504-507, 2006

[5] Marc'Aurelio Ranzato, C. Poultney, S. Chopra, Y. LeCun: "Efficient learning of sparse representations with an energy-based mode". *Advances in Neural Information Processing Systems 19*, p1137–1144, 2007

[6] Xavier Glorot, Y. Bengio: "Understanding the difficulty of training deep feedforward neural networks". *Proceedings of the International Conference on Artificial Intelligence and Statistics. Society for Artificial Intelligence and Statistics.* vol. 9, p249-256, 2010

[7] Xavier Glorot, A. Bordes, Y. Bengio: "Deep sparse rectifier neural networks|. *International Conference on Artificial Intelligence and Statis-tics*, p315–323, 2011

[8] Olga Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, L. Fei-Fei: "ImageNet Large Scale Visual Recognition Challenge". *International Journal of Computer Vision*, vol. 115(3), p211-252, 2015

[9] Tom Le Paine, P. Khorrami, W. Han, T. S. Huang: "An Analysis of Unsupervised Pre-training in Light of Recent Advances". *arXiv preprint.* arXiv: 1412.6597, April 2015

[10] Yann LeCun's answer in reddit posts. Questions and answers available on *https://www.reddit.com/r/MachineLearning/comments/25lnbt/ama_yann_lecun/?limit=* retrieved on 23 October, 2016

[11] Cybenko George: "Approximation by superpositions of a sigmoidal function". *Mathematics of control, signals and systems*, vol. 2(4), p303-314, 1989

[12] Yoshua Bengio, Y. LeCun: "Scaling learning algorithms towards AI". *Large Scale Kernel Machines*, p321–360, MIT Press, 2007

[13] John S. Bridle: "Training Stochastic Model Recognition Algorithms as Networks can Lead to Maximum Mutual Information Estimation of Parameters". *Advances in Neural Information Processing Systems 2 (NIPS)* , p211–217, 1990

[14] Andrew L Maas, A. Y. Hannun, A. Y Ng: "Rectifier nonlinearities improve neural network acoustic models". *Prec ceding of International Conference on Machine Learning*, vol. 30(1), 2013

[15] Kaiming He, X. Zhang, S. Ren, J. Sun: "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification". *The IEEE International Conference on Computer Vision (ICCV)*, p1026-1034, 2015

[16] Djork-Arné Clevert, T. Unterthiner, S. Hochreiter: "Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs)". *International Conference on Learning Representations (ICLR)*, 2016

[17] Kishore Konda, R. Memisevic, D. Krueger: "Zero-bias autoencoders and the benefits of co-adapting features". *International Conference on Learning Representations (ICLR)*, 2015

[18] Xiaojie Jin, C. Xu, J. Feng, Y. Wei, J. Xiong, S. Yan: "Deep Learning with S-shaped Rectified Linear Activation Units". *arXiv preprint*, arXiv:1512.07030, 2015

[19] Yann LeCun, Y. Bengio, G. E. Hinton: "Deep Learning". *Nature*, vol. 521(7553), p436-444, 2015

[20] Warren S. McCulloch, W Pitts: "A logical calculus of the ideas immanent in nervous activity". *Bulletin of Mathematical Biophysics*, vol. 5 (4), p115–133, 1943

[21] Frank Rosenblatt: "The perceptron: A probabilistic model for information storage and organization in the brain". *Psychological Review*, vol. 65(6), p386-408, 1958

[22] Marvin Minsky, S. Papert (1969), *Perceptrons: An Introduction to Computational Geometry*, MIT Press, 1969

[23] Seppo Linnainmaa: "The representation of the cumulative rounding error of an algorithm as a Taylor expansion of the local rounding errors". *Master thesis*, University of Helsinki, 1970

[24] Paul Werbos: "Beyond regression: New tools for prediction and analysis in the behavioral sciences". *PhD thesis*, Harvard University, 1974

[25] Paul Werbos: "Applications of advances in nonlinear sensitivity analysis". *System Modeling and Optimization*, p762-770, 1982

[26] David E. Rumelhart, G. E. Hinton, R. J. Williams: "Learning representations by back-propagating errors". *Nature*, vol. 323, p533-536, 1986

[27] Nico Galoppo, N. Govindaraju, M. Henson, D. Manocha: "LU-GPU: Efficient Algorithms for Solving Dense Linear Systems on Graphics Hardware". *Proceedings of the ACM/IEEE SC 2005 Conference on SuperComputing*, 2005

[28] Nitish Srivastava, G. E. Hinton, A. Krizhevsky, I. Sutskever, R. Salakhutdinov: "Dropout: a simple way to prevent neural networks from overfitting". *Journal of Machine Learning Research*, p1929-1958, 2014

[29] Li Wan, M. Zeiler, S. Zhang, Y. LeCun, R. Fergus: "Regularization of neural networks using dropconnect". *Proceedings of the 30th International Conference on Machine Learning (ICML-13)*, p1058-1066, 2013

[30] John Duchi, E. Hazan, T. Singer: "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization". *Journal of Machine Learning Research*, vol. 12, p2121-2159, 2011

[31] Matthew D. Zeiler: "Adadelta: An Adaptive Learning Rate Method". *arXiv preprint*, arXiv:1212.5701, 2012

[32] Dmytro Mishkin, J. Matas: "All you need is a good init". *International Conference on Learning Representations (ICLR)*, 2016

[33] Karen Simonyan, A. Zisserman: "Very Deep Convolutional Networks for Large-Scale Image Recognition". *International Conference on Learning Representations (ICLR)*, 2015

[34] Kaiming He, X. Zhang, S. Ren, J. Sun: "Deep Residual Learning for Image Recognition". *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016

[35] David H. Hubel, T. N. Wiesel: "Receptive fields and functional architecture of monkey striate cortex". *The Journal of Physiology*, vol. 195(1), p215–243, 1968

[36] Kunihiko Fukuyama: "A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position". *Biological Cybernetics*, vol. 36, p193-202, 1980

[37] Yann LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, L. D. Jackel: "Backpropagation applied to handwritten zip code recognition". *Neural Computation*, vol. 1(4), p541-551, 1989

[38] Alex Krizhevsky, I. Sutskever, G. E. Hinton: "Imagenet classification with deep convolutional neural networks". *Advances in neural information processing systems*, p1097-1105, 2012

[39] Jonathan Long, E. Shelhamer, T. Darrell: "Fully convolutional networks for semantic segmentation". *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, p3431-3440, 2015

[40] Jost Tobias Springenberg, A. Dosovitskiy, T. Brox, M. Riedmiller: "Striving for simplicity: The all convolutional net". *arXiv preprint*, arXiv:1412.6806, 2014

[41] Pascal Vincent, H. Larochelle, I. Lajoie, Y. Bengio, P. A. Manzagol: "Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion. *The Journal of Machine Learning Research*, vol. 11, p3371-3408, 2010

[42] Jonathan Masci, U. Meier, D. Cireşan, Scmidhuber: "Stacked Convolutional Auto-Encoders for Hierarchical Feature Extraction". *Proceedings of the 21th International Conference on Artificial Neural Networks*, vol. 1, p52-59, 2011

[43] Yasi Wang, H. Yao, S. Zhao: "Auto-encoder based dimensionality reduction". *Neurocomputing*, vol. 184, p232–242, 2016

[44] Diederik P Kingma, M. Welling: "Auto-Encoding Variational Bayes". *Proceedings of the 2nd International Conference on Learning Representations (ICLR)*, 2014

[45] Yoshua Bengio, L. Yao, G. Alain, P. Vincent: "Generalized denoising auto-encoders as generative models". *Advances in Neural Information Processing Systems*, p899-907, 2013

[46] Junyuan Xie, L. Xu, E. Chen: "Image Denoising and Inpainting with Deep Neural Networks". *Advances in Neural Information Processing Systems 25 (NIPS)*, 2012

[47] Xugang Lu, Y. Tsao, S. Matsuda, C. Hori: "Speech Enhancement Based on Deep Denoising Autoencoder". *Interspeech*, p436-440, 2013

[48] Jun Deng, R. Xia, Z. Zhang, Y. Liu, B. Schuller: "Introducing shared-hidden-layer autoencoders for transfer learning and their application in acoustic emotion recognition". *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, p4818-4822, 2014

[49] Jun Deng, Z. Zhang, E. Marchi, B. Schuller: "Sparse Autoencoder-Based Feature Transfer Learning for Speech Emotion Recognition". *Humaine Association Conference on Affective Computing and Intelligent Interaction(ACII*, p511-p516, 2013

[50] Fuzhen Zhuang, X. Cheng, P. Luo, S. J. Pan, Q. He: "Supervised Representation Learning: Transfer Learning with Deep Autoencoders". *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligenc*, p4119-4125, 2015

[51] Chenggang Zhang, J. Song, W. Gao, J. Jiang: "An imbalanced data classification algorithm of improved autoencoder neural network". *Eighth International Conference on Advanced Computational Intelligence (ICACI)*, 2016

[52] Zuhe Li, Y. Fan, F. Wang: "Convolutional Autoencoder-based Color Image Classification using Chroma Subsampling in YCbCr Space". *International Congress on Image and Signal Processing (CISP)*, 2015

[53] Stavros Petridis, M. Pantic: "Deep complementary bottleneck features for visual speech recognition". *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, p2304-2308, 2016

[54] Wei Wang, Y. Huang, Y. Wang, L. Wang: "Generalized Autoencoder: A Neural Network Framework for Dimensionality Reduction". *Proceedings of the 2014 IEEE Conference on Computer Vision and Pattern Recognition Workshops*, p496-503 , 2014

[55] Alireza Makhzani, B. Frey: "k-Sparse Autoencoders". *International Conference on Learning Representations (ICLR)*, 2014

[56] Alireza Makhzani, B. J. Frey: "Winner-take-all autoencoders". *Advances in Neural Information Processing Systems*, p2791-2799, 2015

[57] Junbo Zhao, M. Mathieu, R. Goroshin, Y. Lecun: "Stacked what-where auto-encoders". *arXiv preprint*, arXiv:1506.02351, 2015

[58] Zuhe Li, Y Fan, W. Liu: "The effect of whitening transformation on pooling operations in convolutional autoencoders". *EURASIP Journal on Advances in Signal Processing*, doi:10.1186/s13634-015-0222-1, Springer, 2015

[59] Zhouhan Lin, R. Memisevic, K. Konda: "How far can we go without convolution: Improving fully-connected networks". *arXiv preprint*, arXiv:1511.02580, 2015

[60] Tom Fawcett: "An introduction to ROC analysis". *Pattern Recognition Letters*, vol. 27 (8), p861-874 , 2006

[61] George. E. Nasr, E. Badr, C. Joun: "Cross Entropy Error Function in Neural Networks: Forecasting Gasoline Demand". *Proceedings of the Fifteenth International Florida Artificial Intelligence Research Society Conference*, p381-384, 2002

[62] Jolm Herts, A. Krogh, R. G. Palmer (1991), *Introduction To The Theory Of Neural Computation*. United States: Westview Press, ISBN:978-0201515602

[63] Krzysztof C. Kiwiel (2001), "Convergence and efficiency of subgradient methods for quasiconvex minimization". *Mathematical programming*. p1-25, Springer, ISSN: 1436-4646

[64] Léon Bottou: "Large-Scale Machine Learning with Stochastic Gradient Descent". *Proceedings of COMPSTAT'2010*, p177-186, 2010

[65] Léon Bottou (1998): "*Online Learning and Neural Networks*",United Kingdom: Cambridge University Press. ISBN 978-0-521-65263-6

[66] Theano Development Team: "Theano: A Python framework for fast computation of mathematical expressions". *arXiv preprint*. arXiv: 1605.02688, May 2016

[67] Martín Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I Goodfellow, A. Harp, G. Irving, M. Isard, R. Jozefowicz, Y. Jia, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, M. Schuster, R. Monga, S. Moore, D. Murray, C. Olah, J. Shlens, B. Steiner, I Sutskever, K. Talwar, P Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng: "TensorFlow: Large-scale machine learning on heterogeneous systems". *arXiv preprint*. arXiv:1603.04467, March 2016

[68] Yurii Nesterov: "A method of solving a convex programming problem with convergence rate O(1/k2)". *Soviet Mathematics Doklady*, vol. 27(2), p372-376, 1983

[69] Jeffrey Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang, Q. V. Le, A. Y. Ng: "Large Scale Distributed Deep Networks". *Advances in Neural Information Processing Systems 25*, p1232–1240, 2012

[70] Yangqing Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, T. Darrell: "Caffe: Convolutional architecture for fast feature embedding". *Proceedings of the 22nd ACM international conference on Multimedia*, p675-678, 2014

[71] François Chollet: "Keras: Deep Learning library for Theano and TensorFlow", *GitHub repository: https://github.com/fchollet/keras* , documentation available on *https://keras.io* . Retrieved 06. September, 2016

[72] Yann LeCun, L. Bottou, G. Orr and K. Muller: "Efficient BackProp". *Lecture Notes in Computer Science*, Springer, vol.1524, p9-50, 1998

[73] C. C. Tan, C Eswaran: "Reconstruction of handwritten digit images using autoencoder neural networks". *Canadian Conference on Electrical and Computer Engineering*, p465-470, 2008

[74] Chunyan Xu, C. Lu; X. Liang, J. Gao, W. Zheng, T. Wang, S. Yan: "Multi-Loss Regularized Deep Neural Network". *IEEE Transactions on Circuits and Systems for Video Technology*, vol. PP (99), 2015

[75] Yann LeCun, L. Bottou, Y. Bengio, P. Halffner: "Gradient-Based Learning Applied to Document Recognition". *Preceedings of the IEEE.* vol. 88(11), p2278-2324, November 1998

[76] Alex Krizhevsky: "Learning multiple layers of features from tiny images". *Computer Science Department, University of Toronto, Tech. Rep*, 1(4):7, 2009

[77] Chen-Yu Lee, P. Gallagher, Z. Tu: "Generalizing pooling functions in convolutional neural networks: Mixed, gated, and tree". *International Conference on Artificial Intelligence and Statistics*, 2016

[78] Dan Cireşan, U. Meier, J. Schmidhuber: "Multi-column Deep Neural Networks for Image Classification". *Computer Vision and Pattern Recognition (CVPR 2012)*, p3642-3649, 2012

[79] Rupesh Kumar Srivastava, Klaus Greff, Jürgen Schmidhuber: "Training very deep networks". *Advances in neural information processing systems*, p2377-2385, 2015

[80] Antonio Torralba, R.Fergus, W.T. Freeman: "80 Million Tiny Images: A Large Data Set for Nonparametric Object and Scene Recognition". *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol.30(11), p958-1970, 2008

[81] Steve Lawrence, C. L. Giles, A. C. Tsoi: "Lessons in Neural Network Training: Overfitting May be Harder than Expected". *Proceedings of the Fourteenth National Conference on Artificial Intelligence*, United States, p540-545, 1997

[82] Andrew Yao: "Separating the polynomial-time hierarchy by oracles". *roceedings of the 26th Annual IEEE Symposium on Foundations of Computer Science*, p1-10, 1985

[83] Johan Håstad, M. Goldmann: "On the power of small-depth threshold circuits". *Computational Complexity*, vol. 1, p113-129, 1991

A. EXAMPLE OF BUILDING, TRAINING AND EVALUATING ANNS WITH KERAS

```
#to the very beginners:
#anything in a line after '#' is for human reading
#and will be ignored (e.g. they are not parts of the code)
#first, we need to load needed modules
from __future__ import print_function
from   keras.datasets import mnist
from   keras.engine.topology import Input
from   keras.layers.core import Activation, Dense, Flatten
from   keras.layers.convolutional import Convolution2D
from   keras.layers.pooling import MaxPooling2D
from   keras.models import Model
from   keras.utils.np_utils import to_categorical
import numpy as np

#main function for this script
def main():
    #load data from built-in mnist data set
    (X_train, y_train), (X_test, y_test) = mnist.load_data()
    nb_class = y_train.max() + 1 #number of classes
    #images saved in X_train and Y_train have size 28*28, we want it to be 1*28*28
    #and we need them to be presented in float numbers
    X_train = X_train[:, np.newaxis, ...].astype(np.float32)
    X_test = X_test[:, np.newaxis, ...].astype(np.float32)
    #convert scale labels into one-hot vectors
    Y_train = to_categorical(y_train, nb_class)
    Y_test = to_categorical(y_test, nb_class)

    #build network
    in_data = Input(shape=(X_train.shape[1:])) #specify input shape
    x = Convolution2D(32, 3, 3)(in_data) # apply convolution with 32
    #3x3 convolutional kernels
    x = Activation('relu')(x) #use ReLU as its activation function
    x = MaxPooling2D((3, 3), strides=(2, 2))(x) #max-pooling with 3x3 window
    #and stride 2 along both axes
    x = Flatten()(x) #flatten feature maps into a long vector
    x = Dense(300)(x) # fully-connected layer with 300 nodes
    x = Activation('relu')(x) # ReLU activation
    x = Dense(nb_class)(x) #fully-connected layer with nb_class
    #(e.g. 10 in this example) nodes
    out_prob = Activation('softmax')(x) #final output
    model = Model(input=in_data, output=out_prob) #build model

    #training
    #a model needs to be compiled before training
    model.compile(optimizer='sgd', loss='categorical_crossentropy', metrics=['accuracy'])
    #train the network with training data for 50 epochs
    model.fit(X_train, Y_train, nb_epoch=50)
```

```
    #after training, the model can be evaluated with model.evaluate()
    #loss and metrics specified in model.compile() function
    #will be saved to the variable score
    #here, verbose=0 means we do not want the text progress bar
    score = model.evaluate(X_test, Y_test, verbose=0)
    print("\nAccuracy on test set is:", score[1])

if __name__ == '__main__':
        main()
```

B. CROSS-ENTROPY ON TRAINING AND TESTING SET DURING OPTIMIZATION

First network trained on MNIST

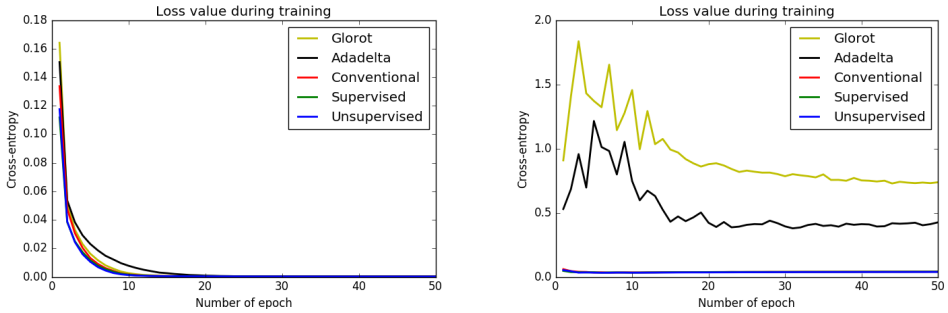

Figure B.1: Cross-entropy on training and test set for First network trained on MNIST. Although the training loss seems to be similar for each other, test loss for three pre-trained classifiers (overlapped below the black curve) is always smaller than the randomly initialized ones.

Second network trained on MNIST

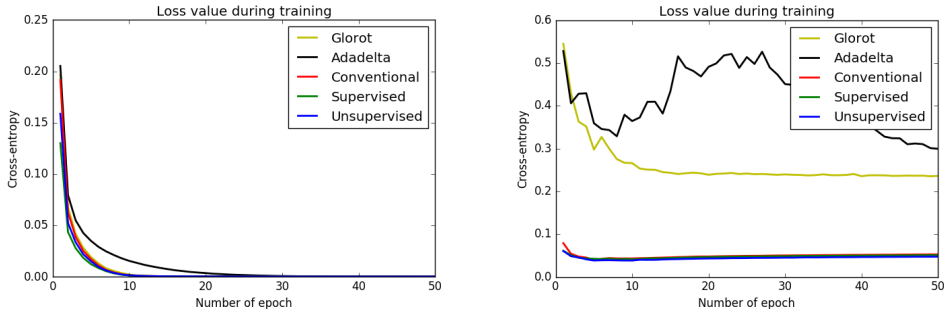

Figure B.2: Cross-entropy on training and test set for second network trained on MNIST. Again, pre-trained classifiers show lower test loss.

Third network trained on MNIST

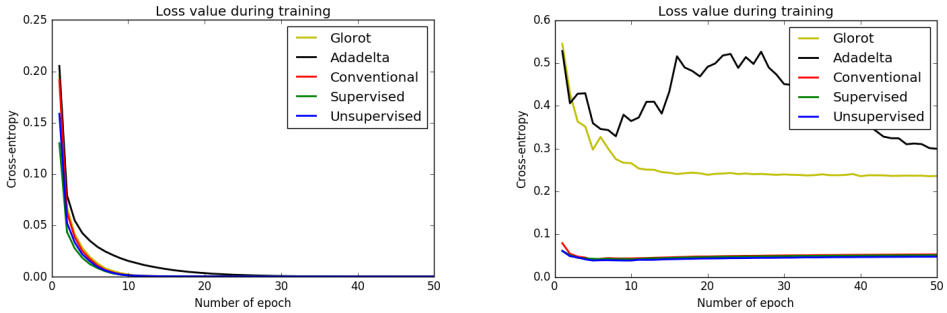

Figure B.3: Cross-entropy on training and test set for third network trained on MNIST. Still, pre-trained classifiers show lower test loss.

First network trained on CIFAR10, without data augmentation

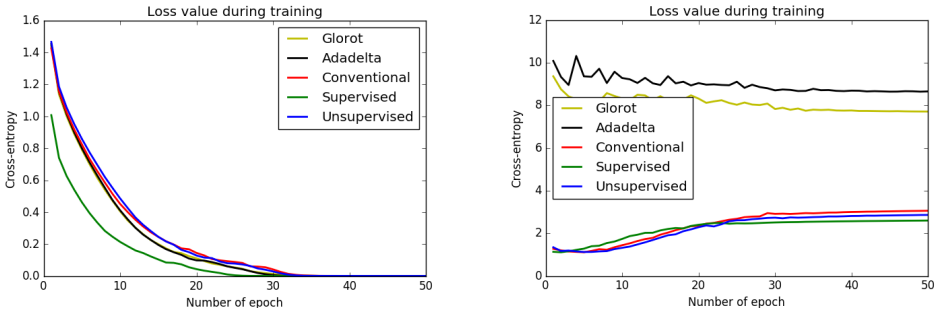

Figure B.4: Cross-entropy on training and test set during training. Pre-trained classifiers show lower test loss but begin to over-fit quickly.

First network trained on CIFAR10, with data augmentation

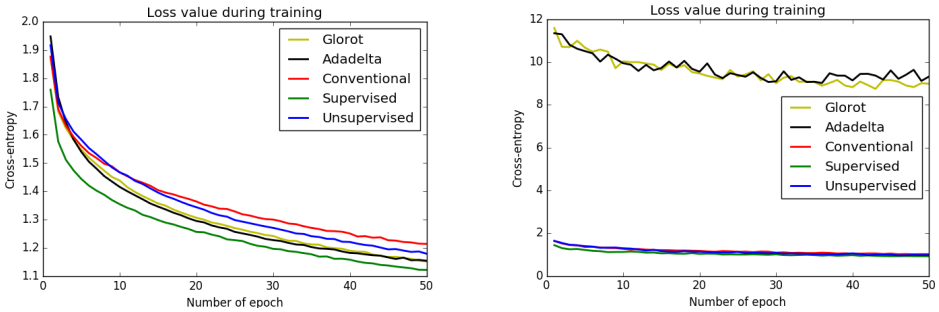

Figure B.5: Cross-entropy on training and test set during training. Pre-trained classifiers show lower test loss and do not over-fit thanks to data augmentation techniques.

Second network trained on CIFAR10, without data augmentation

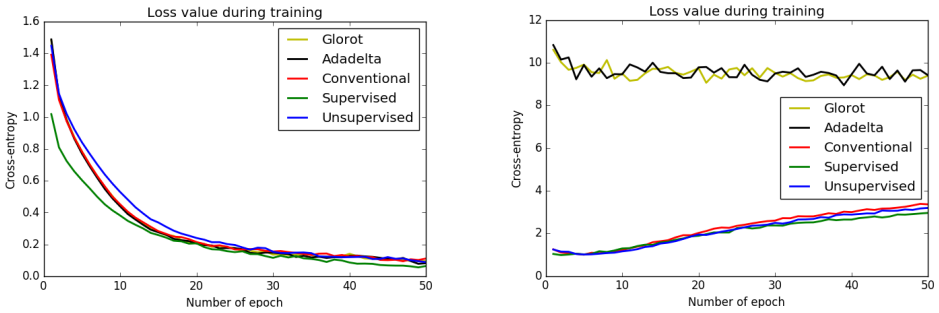

Figure B.6: Cross-entropy on training and test set during training. Pre-trained classifiers show lower test loss but begin to over-fit quickly.

Second network trained on CIFAR10, with data augmentation

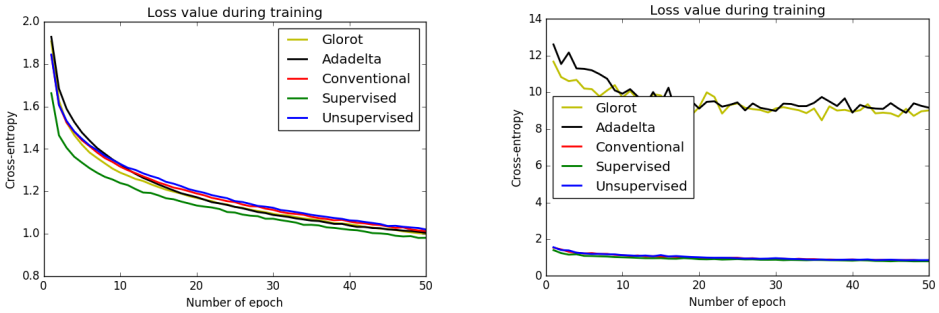

Figure B.7: Cross-entropy on training and test set during training. Pre-trained classifiers show lower test loss and do not over-fit thanks to data augmentation techniques.

Third network trained on CIFAR10, without data augmentation

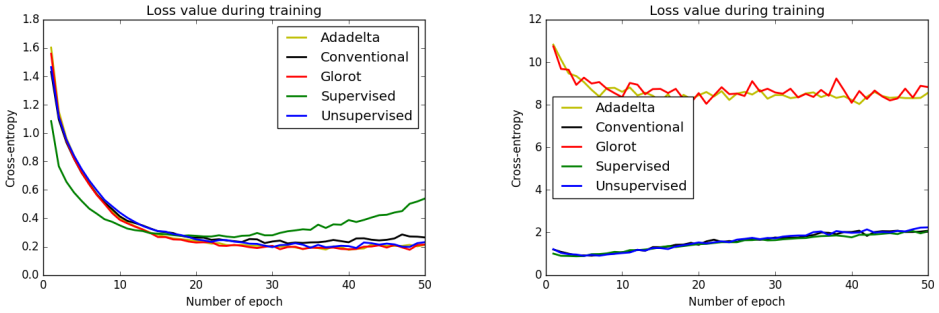

Figure B.8: Cross-entropy on training and test set during training. Pre-trained classifiers show lower test loss but begin to over-fit quickly.

Third network trained on CIFAR10, with data augmentation

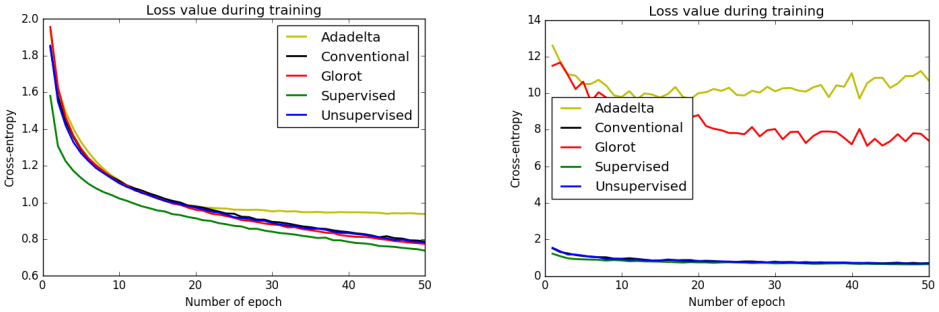

Figure B.9: Cross-entropy on training and test set during training. Pre-trained classifiers show lower test loss and do not over-fit thanks to data augmentation techniques.

First network trained on CIFAR100, without DA

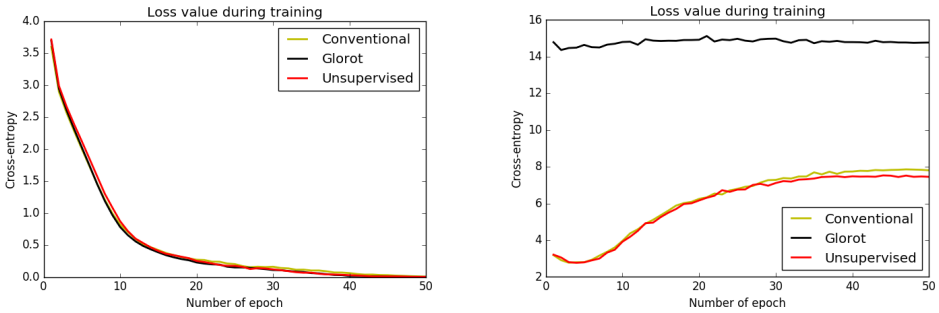

Figure B.10: Cross-entropy on training and test set during training. Pre-trained classifiers show lower test loss but begin to over-fit quickly.

First network trained on CIFAR100, with DA

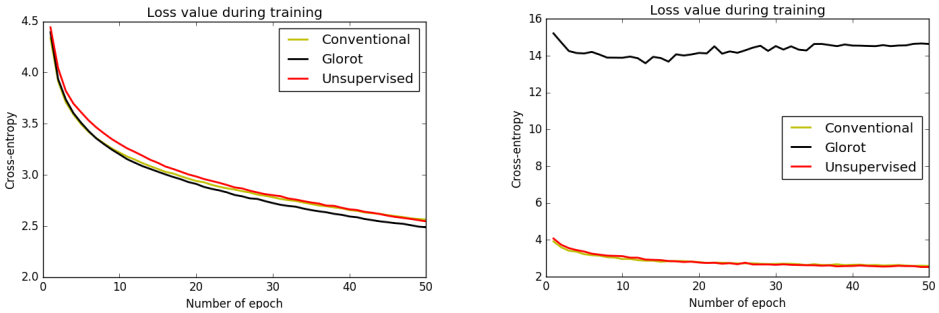

Figure B.11: Cross-entropy on training and test set during training. Pre-trained classifiers show lower test loss and do not over-fit thanks to data augmentation techniques.

Second network trained on CIFAR100, without DA

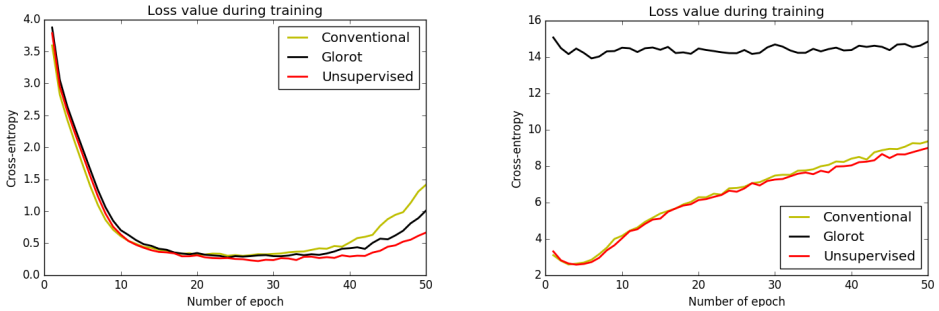

Figure B.12: Cross-entropy on training and test set during training. Pre-trained classifiers show lower test loss but begin to over-fit quickly.

Second network trained on CIFAR100, with DA

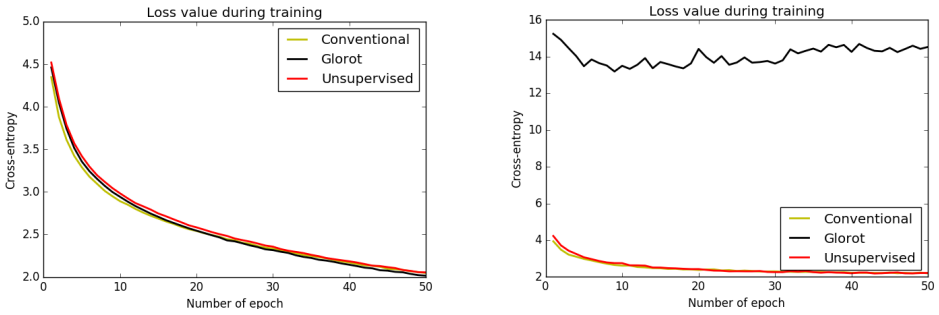

Figure B.13: Cross-entropy on training and test set during training. Pre-trained classifiers show lower test loss and do not over-fit thanks to data augmentation techniques.

C. NETWORK TOPOLOGY USED IN THIS WORK

[**Note**] For all the figures below, the word "Dense" means a fully-connected layer. "input" and "output" specifies input and output diemnsionality of this layer respectively.

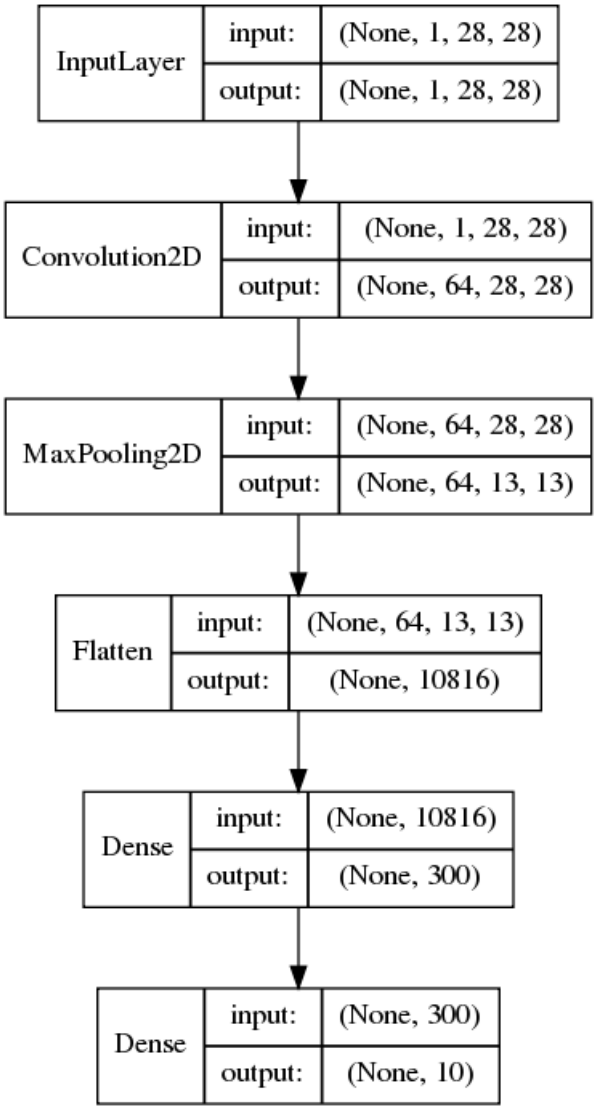

Figure C.1: First network trained on MNIST.

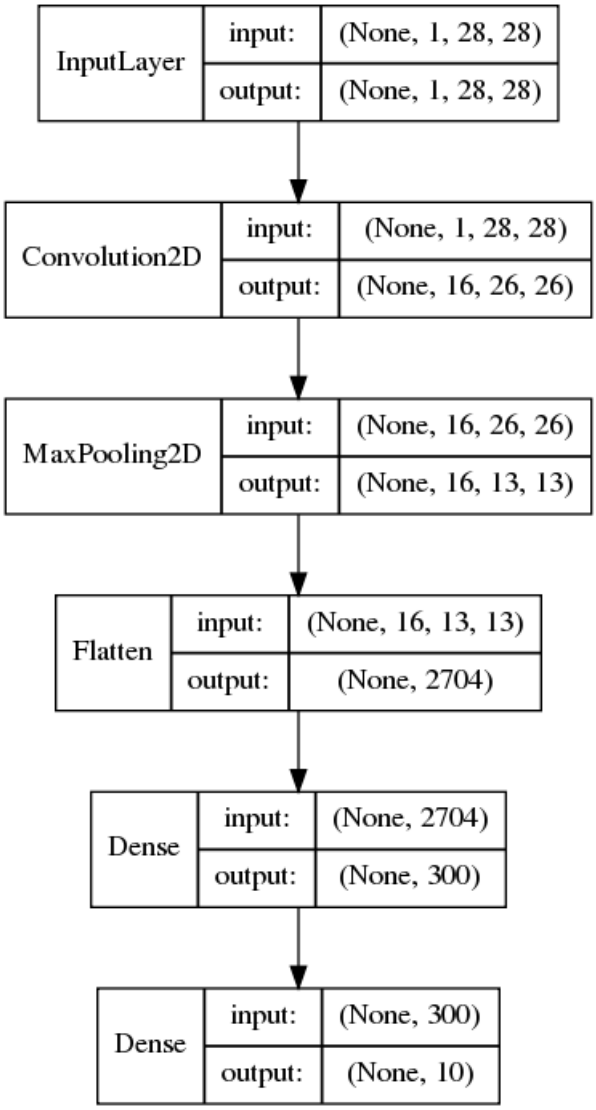

Figure C.2: Second network trained on MNIST.

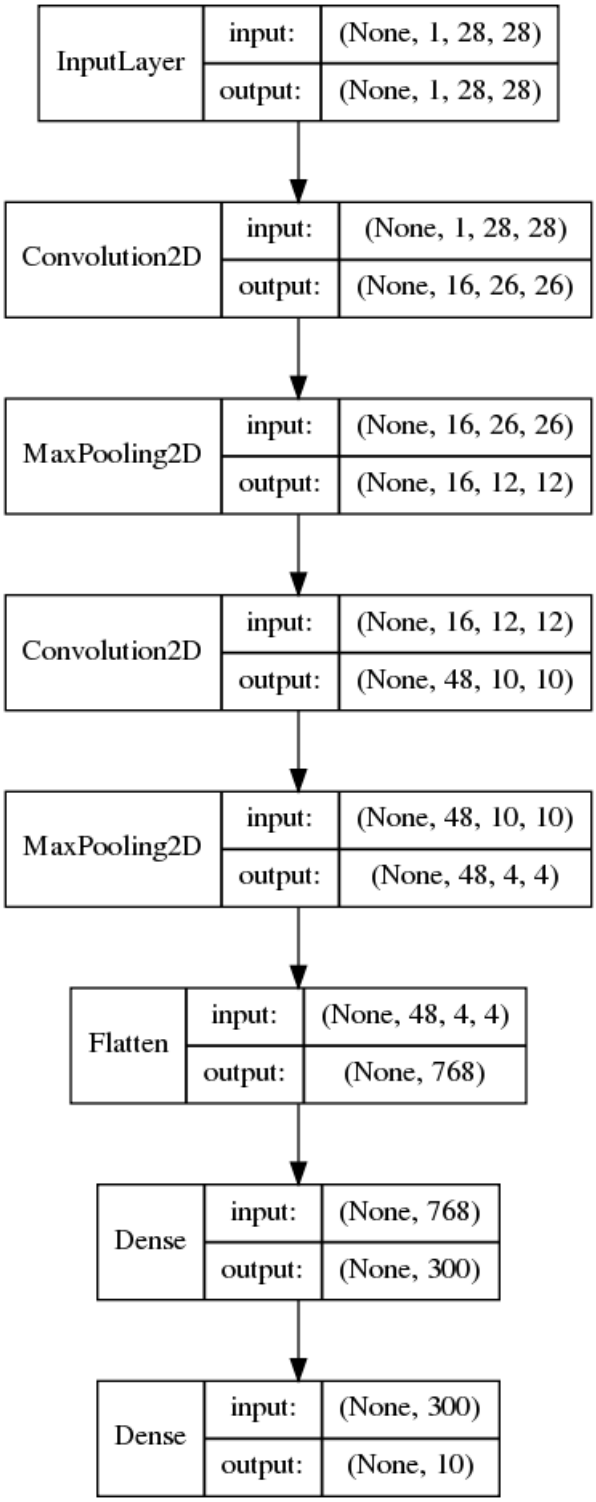

Figure C.3: Third network trained on MNIST.

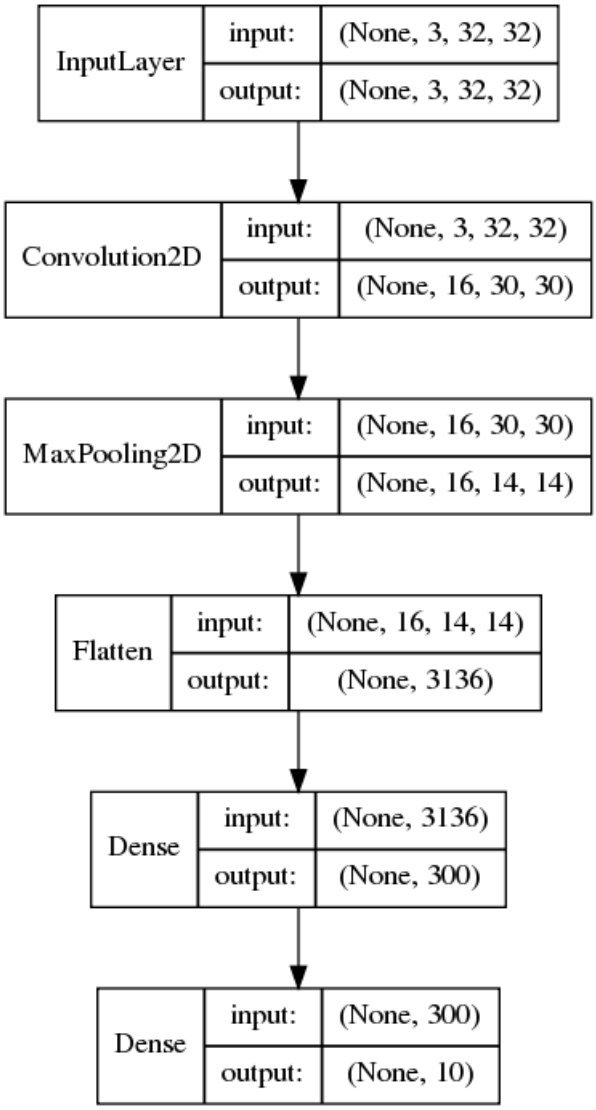

Figure C.4: First network trained on CIFAR10.

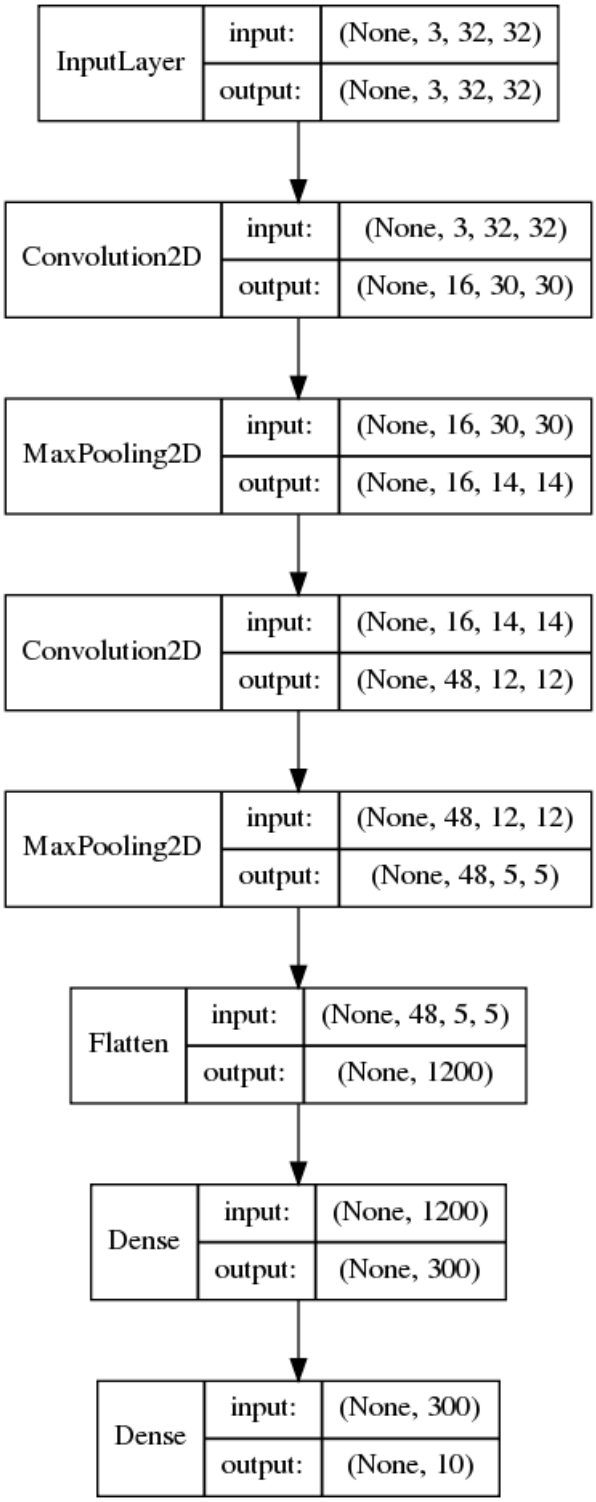

Figure C.5: Second network trained on CIFAR10.

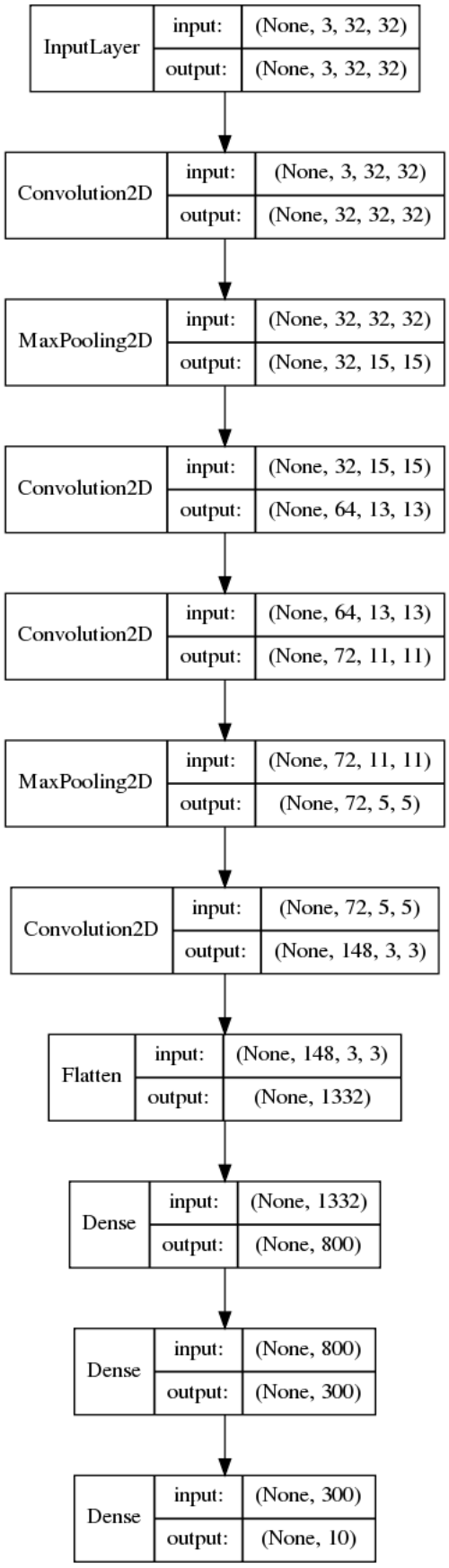

Figure C.6: Third network trained on CIFAR10.

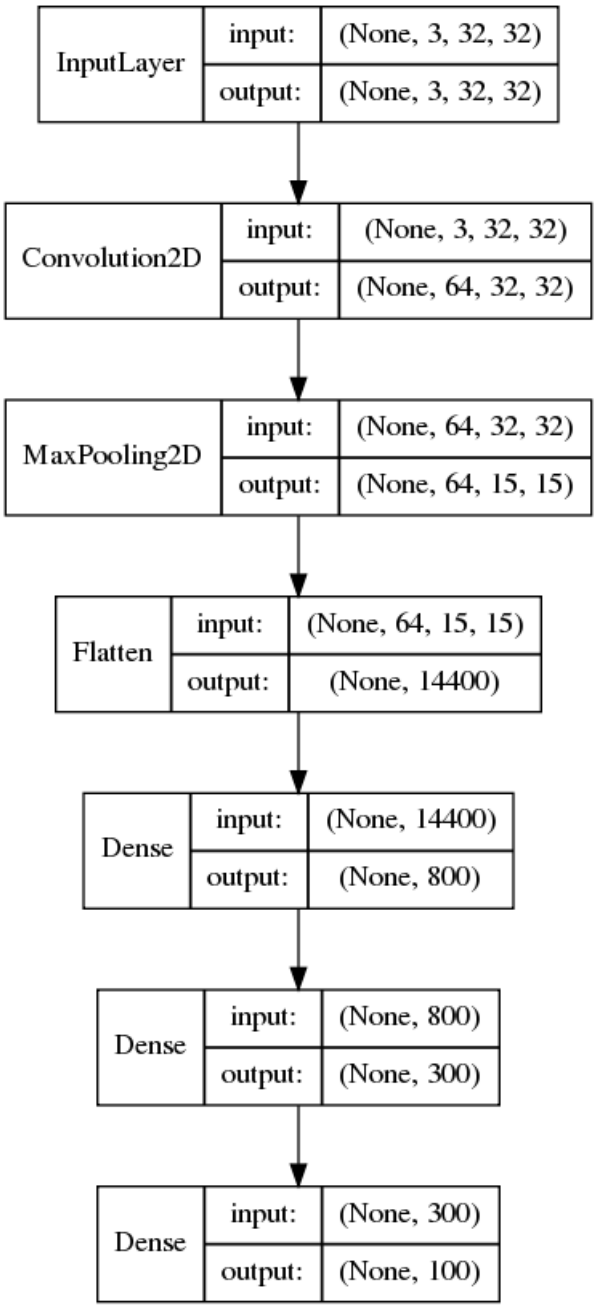

Figure C.7: First network trained on CIFAR100.

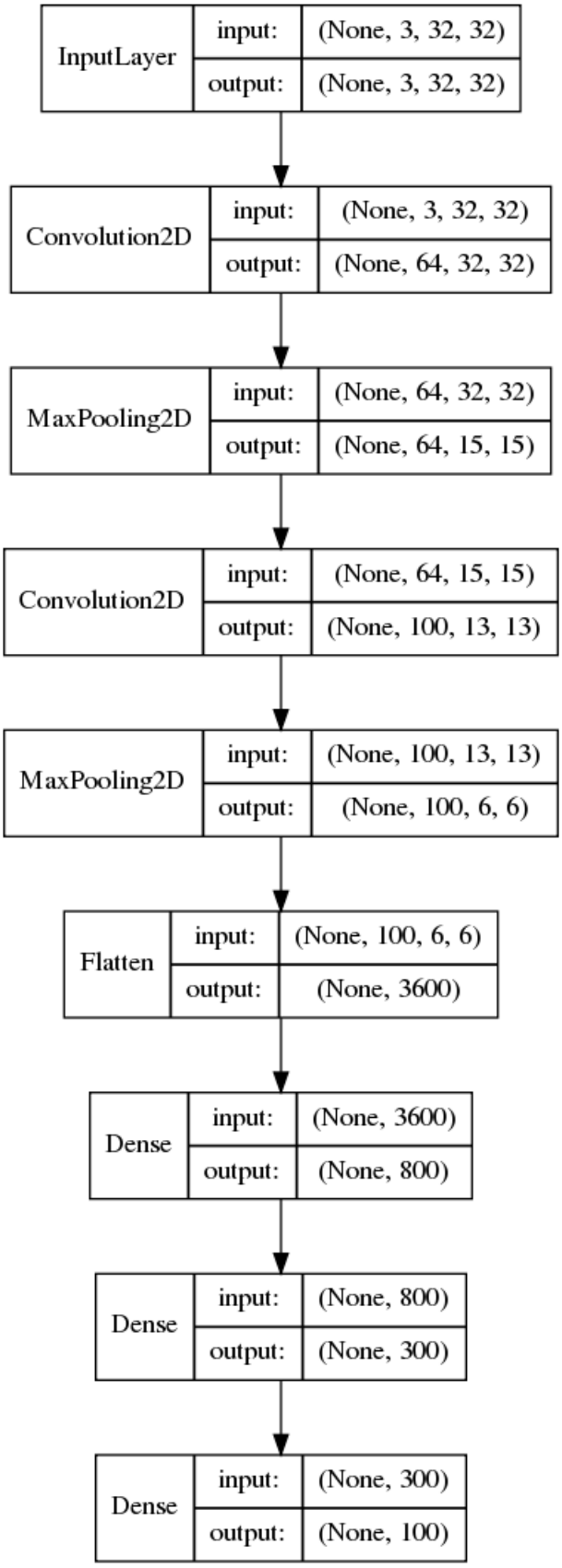

Figure C.8: Second network trained on CIFAR100.

D. NUMBER OF CONTRIBUTORS/STARS ON GITHUB FOR DIFFERENT FRAMEWORKS

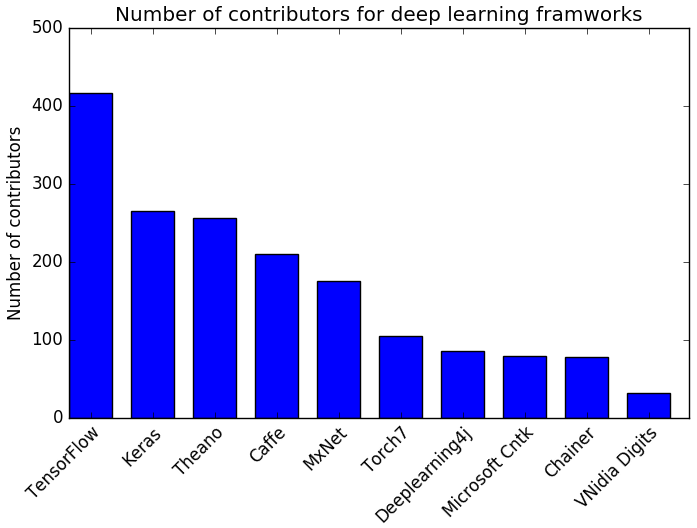

Figure D.1: Number of contributors, till 25^{th} September, 2016

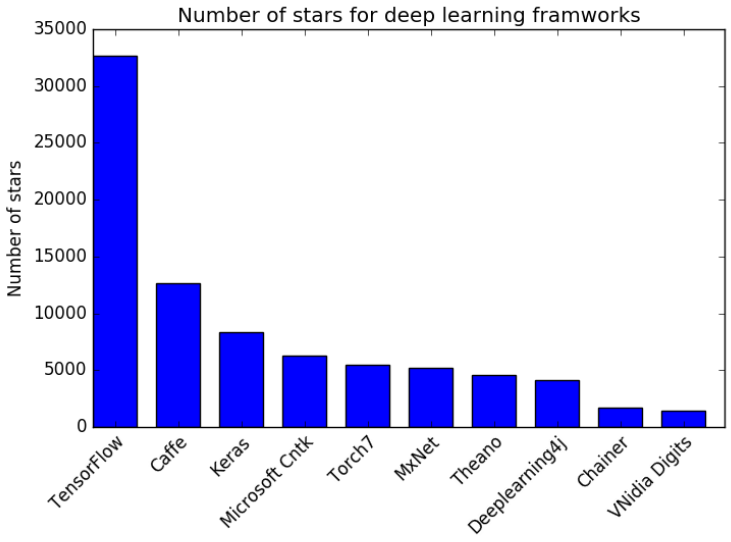

Figure D.2: Number of stars, till 25^{th} September, 2016